VISIT…

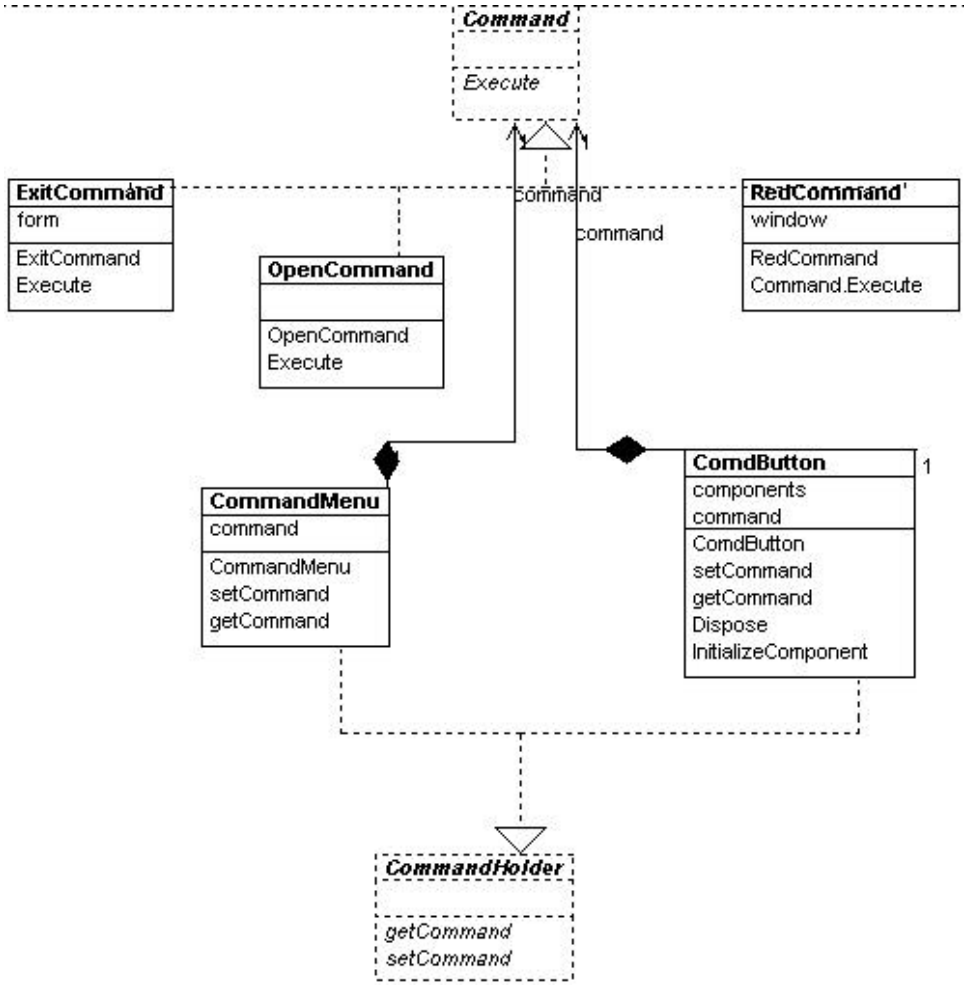


Figure 22-4 – Class diagram of CommandHolder appraoch

Providing Undo

Another of the main reasons for using Command design patterns is that they provide a convenient way to store and execute an Undo function. Each command object can remember what it just did and restore that state

Copyright © , 2002 by James W Cooper

when requested to do so if the computational and memory requirements are not too overwhelming. At the top level, we simply redefine the Command interface to have three methods.

```
public interface Command      {
      void Execute();
      void Undo();
      bool isUndo();
}
```

Then we have to design each command object to keep a record of what it last did so it can undo it. This can be a little more complicated than it first appears, since having a number of interleaved Commands being executed and then undone can lead to some hysteresis. In addition, each command will need to store enough information about each execution of the command that it can know what specifically has to be undone.

The problem of undoing commands is actually a multipart problem. First, you must keep a list of the commands that have been executed, and second, each command has to keep a list of its executions. To illustrate how we use the Command pattern to carry out undo operations, let's consider the program shown in Figure 22-5 that draws successive red or blue lines on the screen, using two buttons to draw a new instance of each line. You can undo the last line you drew with the undo button.

Copyright © , 2002 by James W Cooper

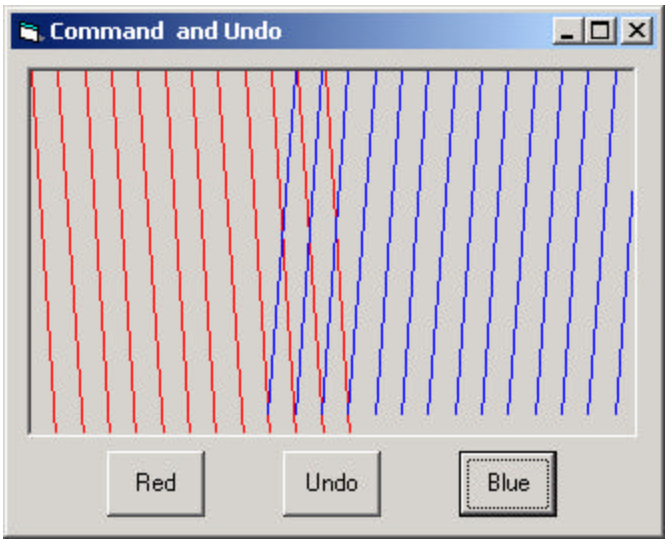


Figure 22-5 – A program that draws red and blue lines each time you click the Red and Blue buttons

If you click on Undo several times, you'd expect the last several lines to disappear no matter what order the buttons were clicked in, as shown in Figure 22-6.

Copyright © , 2002 by James W Cooper

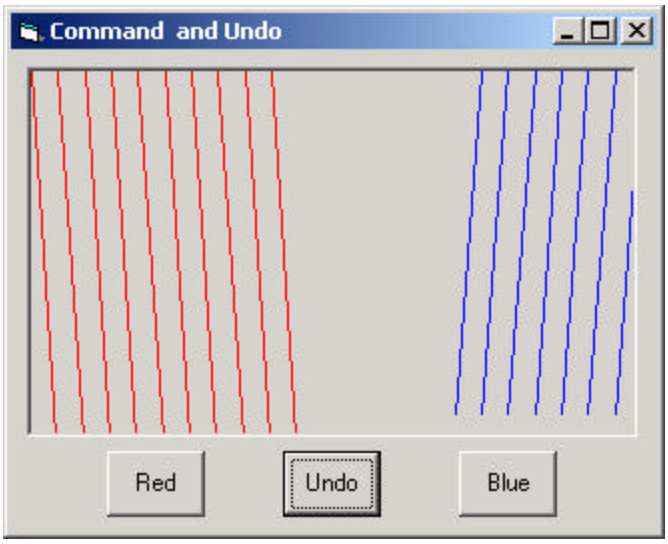


Figure 22-6– The same program as in Figure 22-5 after the Undo button has been clicked several times

Thus, any undoable program needs a single sequential list of all the commands that have been executed. Each time we click on any button, we add its corresponding command to the list.

```
private void commandClick(object sender, EventArgs e) {
       //get the command
       Command comd = ((CommandHolder)sender).getCommand ();
       undoC.add (comd);    //add to undo list
       comd.Execute ();     //and execute it
}
```

Further, the list to which we add the Command objects is maintained inside the Undo command object so it can access that list conveniently.

```
public class UndoComd:Command     {
       private ArrayList undoList;
       public UndoComd()   {
              undoList = new ArrayList ();
       }
       //-----
       public void add(Command comd) {
```

Copyright © , 2002 by James W Cooper

```
            if(! comd.isUndo ()) {
                  undoList.Add (comd);
            }
      }
      //-----
      public bool isUndo() {
            return true;
      }
      //-----
      public void Undo() {             }
      //-----
      public void Execute() {
            int index = undoList.Count - 1;
            if (index >= 0) {
                  Command cmd = (Command)undoList[index];
                  cmd.Undo();
                  undoList.RemoveAt(index);
            }
      }
}
```

The undoCommand object keeps a list of *Commands*, not a list of actual data. Each command object has its unDo method called to execute the actual undo operation. Note that since the undoCommand object implements the Command interface, it, too, needs to have an unDo method. However, the idea of undoing successive unDo operations is a little complex for this simple example program. Consequently, you should note that the *add* method adds all Commands to the list *except* the undoCommand itself, since we have just defined undoing an unDo command as doing nothing. For this reason, our new Command interface includes an isUndo method that returns false for the RedCommand and BlueCommand objects and true for the UndoCommand object.

The redCommand and blueCommand classes simply use different colors and start at opposite sides of the window, although both implement the revised Command interface. Each class keeps a list of lines to be drawn in a Collection as a series of D*rawData* objects containing the coordinates of each line. Undoing a line from either the red or the blue line list simply means removing the last D*rawData* object from the *drawList* collection.

Copyright © , 2002 by James W Cooper

Then either command forces a repaint of the screen. Here is the BlueCommand class.

```
public class BlueCommand :Command       {
       protected Color color;
       private PictureBox pbox;
       private ArrayList drawList;
       protected int x, y, dx, dy;
//-----
       public BlueCommand(PictureBox pbx) {
              pbox = pbx;
              color = Color.Blue ;
              drawList = new ArrayList ();
              x = pbox.Width ;
              dx = -20;
              y = 0;
              dy = 0;
       }
       //-----
       public void Execute() {
             DrawData dl = new DrawData(x, y, dx, dy);
              drawList.Add(dl);
              x = x + dx;
              y = y + dy;
              pbox.Refresh();
       }
       //-----
       public bool isUndo() {
              return false;
       }
       //-----
       public void Undo() {
              DrawData dl;
              int index = drawList.Count - 1;
              if (index >= 0) {
                     dl = (DrawData)drawList[index];
                     drawList.RemoveAt(index);
                     x = dl.getX();
                     y = dl.getY();
              }
              pbox.Refresh();
       }
       //-----
       public void draw(Graphics g) {
              Pen rpen = new Pen(color, 1);
```

Copyright © , 2002 by James W Cooper

```
            int h = pbox.Height;
            int w = pbox.Width;
                  for (int i = 0; i < drawList.Count ; i++) {
                  DrawData dl = (DrawData)drawList[i];
                  g.DrawLine(rpen, dl.getX(), dl.getY(),
                        dl.getX() + dx, dl.getDy() + h);
            }
      }
}
```

Note that the *draw* method in the *drawCommand* class redraws the entire list of lines the command object has stored. These two draw methods are called from the paint handler of the form.

```
public void paintHandler(object sender, PaintEventArgs e) {
      Graphics g = e.Graphics ;
      blueC.draw(g);
      redC.draw (g);
}
```

We can create the RedCommand in just a few lines by deriving from the BlueCommand:

```
public class RedCommand : BlueCommand   {
      public RedCommand(PictureBox pict):base(pict)        {
            color = Color.Red;
            x = 0;
            dx = 20;
            y = 0;
            dy = 0;
      }
}
```

The set of classes we use in this Undo program is shown in Figure 22-7

Copyright © , 2002 by James W Cooper

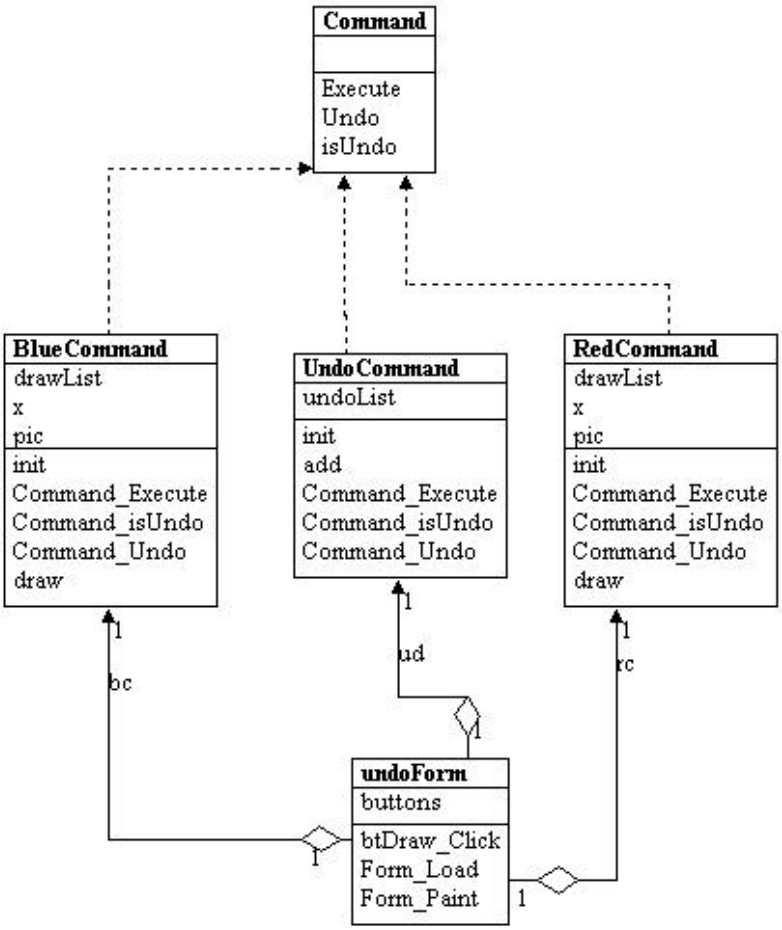


Figure 22-7– The classes used to implement Undo in a Command pattern implementation

Copyright © , 2002 by James W Cooper

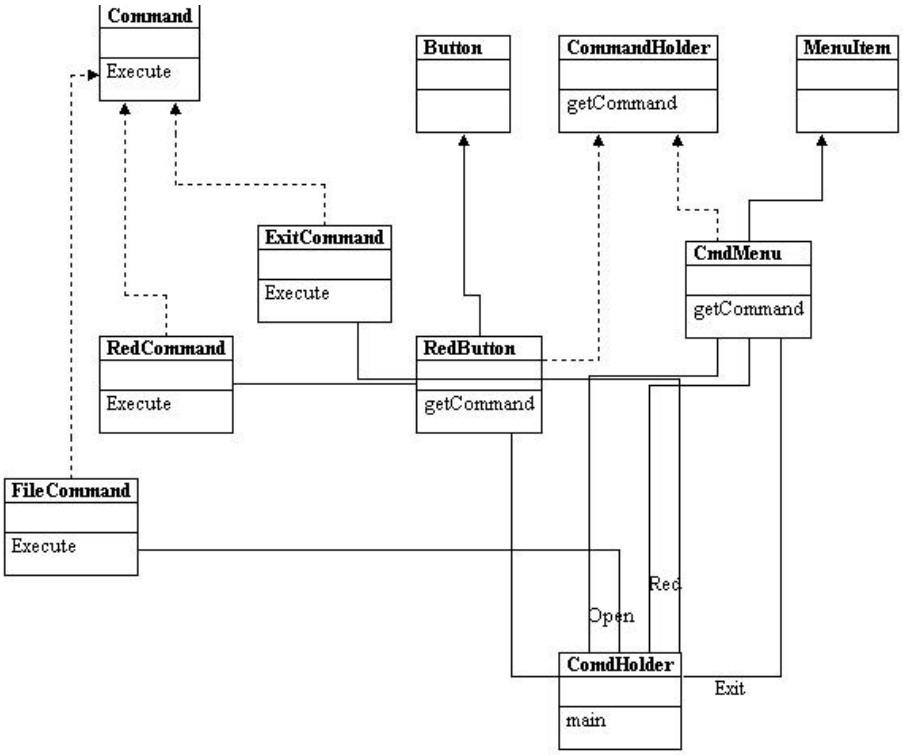


Figure 22-8– A class structure for three different objects that all implement the Command interface and two that implement the CommandHolder interface

Thought Questions

1. Mouse clicks on list box items and on radio buttons also constitute commands. Clicks on multiselect list boxes could also be represented as commands. Design a program including these features.
2. A lottery system uses a random number generator constrained to integers between 1 and 50. The selections are made at intervals selected by a random timer. Each selection must be unique. Design command patterns to choose the winning numbers each week.

Copyright © , 2002 by James W Cooper

Programs on the CD-ROM

\Command\ButtonMenu	Buttons and menus using Command pattern
\Command\UndoCommand	C# program showing line drawing and Undo
\Command\ComdHolder	C# program showing CommandHolder interface

Copyright © , 2002 by James W Cooper

23. The Interpreter Pattern

Some programs benefit from having a language to describe operations they can perform. The Interpreter pattern generally describes defining a grammar for that language and using that grammar to interpret statements in that language.

Motivation

When a program presents a number of different but somewhat similar cases it can deal with, it can be advantageous to use a simple language to describe these cases and then have the program interpret that language. Such cases can be as simple as the sort of Macro language recording facilities a number of office suite programs provide or as complex as Visual Basic for Applications (VBA). VBA is not only included in Microsoft Office products, but it can be embedded in any number of third-party products quite simply.

One of the problems we must deal with is how to recognize when a language can be helpful. The Macro language recorder simply records menu and keystroke operations for later playback and just barely qualifies as a language; it may not actually have a written form or grammar. Languages such as VBA, on the other hand, are quite complex, but they are far beyond the capabilities of the individual application developer. Further, embedding commercial languages usually require substantial licensing fees, which makes them less attractive to all but the largest developers.

Applicability

As the *SmallTalk Companion* notes, recognizing cases where an Interpreter can be helpful is much of the problem, and programmers without formal language/compiler training frequently overlook this

Copyright © , 2002 by James W Cooper

approach. There are not large numbers of such cases, but there are three general places where languages are applicable.

1. *When you need a command interpreter to parse user commands*. The user can type queries of various kinds and obtain a variety of answers.
2. *When the program must parse an algebraic string.* This case is fairly obvious. The program is asked to carry out its operations based on a computation where the user enters an equation of some sort. This frequently occurs in mathematical-graphics programs where the program renders a curve or surface based on any equation it can evaluate. Programs like *Mathematica* and graph drawing packages such as *Origin* work in this way.
3. *When the program must produce varying kinds of output.* This case is a little less obvious but far more useful. Consider a program that can display columns of data in any order and sort them in various ways. These programs are frequently referred to as Report Generators, and while the underlying data may be stored in a relational database, the user interface to the report program is usually much simpler than the SQL language that the database uses. In fact, in some cases, the simple report language may be interpreted by the report program and translated into SQL.

A Simple Report Example

Let's consider a simplified report generator that can operate on five columns of data in a table and return various reports on these data. Suppose we have the following results from a swimming competition.

```
Amanda McCarthy             12  WCA         29.28
Jamie Falco                 12  HNHS        29.80
Meaghan O'Donnell           12  EDST        30.00
Greer Gibbs                 12  CDEV        30.04
Rhiannon Jeffrey            11  WYW         30.04
Sophie Connolly             12  WAC         30.05
Dana Helyer                 12  ARAC        30.18
```

Copyright © , 2002 by James W Cooper

The five columns are *frname, lname, age, club* and *time.* If we consider the complete race results of 51 swimmers, we realize that it might be convenient to sort these results by club, by last name, or by age. Since there are a number of useful reports we could produce from these data in which the order of the columns changes as well as the sorting, a language is one useful way to handle these reports.

We'll define a very simple nonrecursive grammar of this sort.

```
Print lname frname club time Sortby club Thenby time
```

For the purposes of this example, we define these three verbs.

```
Print
Sortby
Thenby
```

And we'll define the five column names we listed earlier.

```
Frname
Lname
Age
Club
Time
```

For convenience, we'll assume that the language is case insensitive. We'll also note that the simple grammar of this language is punctuation free and amounts in brief to the following.

Print var[var] [sortby var [thenby var]]

Finally, there is only one main verb, and while each statement is a declaration, there is no assignment statement or computational ability in this grammar.

Copyright © , 2002 by James W Cooper

Interpreting the Language

Interpreting the language takes place in three steps.

1. Parsing the language symbols into tokens.
2. Reducing the tokens into actions.
3. Executing the actions.

We parse the language into tokens by simply scanning each statement with a StringTokenizer and then substituting a number for each word. Usually parsers push each parsed token onto a *stack* we will use that technique here. We implement the Stack class using an Arraylist—where we have *push, pop, top,* and *nextTop* methods to examine and manipulate the stack contents.

After parsing, our stack could look like this.

Type	Token	
Var	Time	<-top of stack
Verb	Thenby	
Var	Club	
Verb	Sortby	
Var	Time	
Var	Club	
Var	Frname	
verb	Lname	

However, we quickly realize that the "verb" *Thenby* has no real meaning other than clarification, and it is more likely that we'd parse the tokens and skip the *Thenby* word altogether. Our initial stack then, looks like this.

```
Time
Club
Sortby
Time
Club
Frname
```

Copyright © , 2002 by James W Cooper

```
Lname
Print
```

Objects Used in Parsing

In this parsing procedure, we do not push just a numeric token onto the stack but a *ParseObject* that has the both a type and a value property.

```
public class ParseObject   {
       public const int VERB=1000;
       public const int VAR=1010;
       public const int MULTVAR=1020;
       protected int value, type;
       //-----
       public ParseObject(int val, int typ)    {
              value = val;
              type = typ;
       }
       //-----
       public int getValue() {
              return value;
       }
       //-----
       public int getType() {
              return type;
       }
}
```

These objects can take on the type VERB or VAR. Then we extend this object into ParseVerb and ParseVar objects, whose value fields can take on PRINT or SORT for ParseVerb and FRNAME, LNAME, and so on for ParseVar. For later use in reducing the parse list, we then derive *Print* and *Sort* objects from ParseVerb.

This gives us a simple hierarchy shown in Figure 23-1

Copyright © , 2002 by James W Cooper

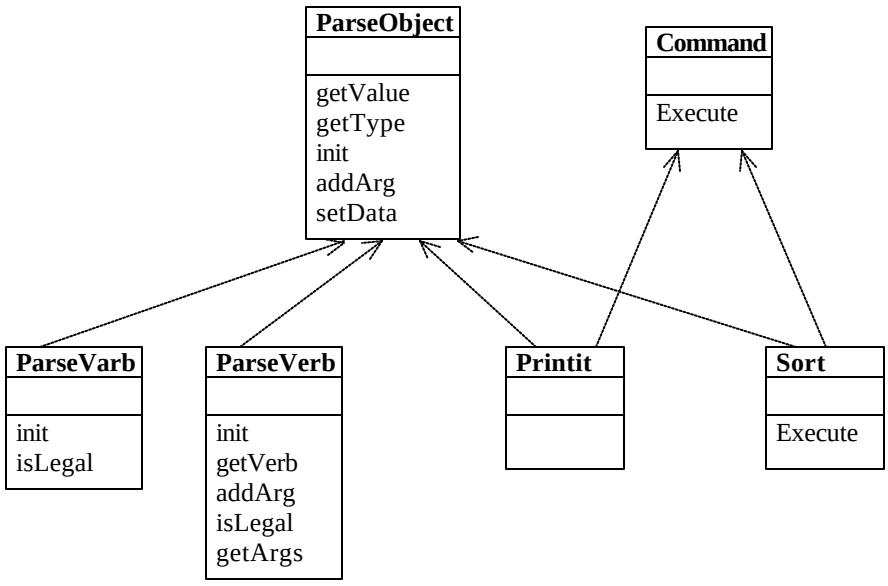


Figure 23-1– A simple parsing hierarchy for the Interpreter pattern

The parsing process is just the following simple code, using the StringTokenizer and the parse objects. Part of the main Parser class is shown here.

```
public class Parser        {
       private Stack stk;
       private ArrayList actionList;
       private Data dat;
       private ListBox ptable;
       private Chain chn;
       //-----
       public Parser(string line, KidData kd, ListBox pt)    {
              stk = new Stack ();
              //list of verbs accumulates here
              actionList = new ArrayList ();
              setData(kd, pt);
              buildStack(line);    //create token stack
              buildChain();        //create chain of responsibility
       }
```

Copyright © , 2002 by James W Cooper

```
//-----
private void buildChain() {
       chn = new VarVarParse(); //start of chain
       VarMultvarParse vmvp = new VarMultvarParse();
       chn.addToChain(vmvp);
       MultVarVarParse mvvp = new MultVarVarParse();
       vmvp.addToChain(mvvp);
       VerbMultvarParse vrvp = new VerbMultvarParse();
       mvvp.addToChain(vrvp);
       VerbVarParse vvp = new VerbVarParse();
       vrvp.addToChain(vvp);
       VerbAction va = new VerbAction(actionList);
       vvp.addToChain(va);
       Nomatch nom = new Nomatch ();     //error handler
       va.addToChain (nom);
}
//-----
public void setData(KidData kd, ListBox pt) {
       dat = new Data(kd.getData ());
       ptable = pt;
}
//-----
private void buildStack(string s) {
       StringTokenizer tok = new StringTokenizer (s);
       while(tok.hasMoreElements () ) {
              ParseObject token = tokenize(tok.nextToken ));
                     stk.push (token);                }
}
//-----
protected ParseObject tokenize(string s) {
       ParseObject obj;
       int type;
       try {
              obj = getVerb(s);
              type = obj.getType ();
       }
       catch(NullReferenceException) {
              obj = getVar(s);
       }
       return obj;
}
//-----
protected ParseVerb getVerb(string s) {
       ParseVerb v = new ParseVerb (s, dat, ptable);
       if(v.isLegal () )
```

Copyright © , 2002 by James W Cooper

```
                return v.getVerb (s);
        else
                return null;
    }
    //-----
    protected ParseVar getVar(string s) {
        ParseVar v = new ParseVar (s);
        if( v.isLegal())
                return v;
        else
                return null;
    }
}
```

The ParseVerb and ParseVar classes return objects with *isLegal* set to true if they recognize the word.

```
public class ParseVerb:ParseObject      {
    protected const int PRINT = 100;
    protected const int SORT = 110;
    protected const int THENBY = 120;
    protected ArrayList args;
    protected Data kid;
    protected ListBox pt;
    protected ParseVerb pv;
    //-----
    public ParseVerb(string s, Data kd, ListBox ls):
                base(-1, VERB) {
        args = new ArrayList ();
        kid = kd;
        pt = ls;
        if(s.ToLower().Equals ("print")) {
                value = PRINT;
        }
        if(s.ToLower().Equals ("sortby")) {
                value = SORT;
        }
    }
    //------
    public ParseVerb getVerb(string s) {
        pv = null;
        if(s.ToLower ().Equals ("print"))
                pv =new Print(s,kid, pt);
        if(s.ToLower ().Equals ("sortby"))
                pv = new Sort (s, kid, pt);
```

Copyright © , 2002 by James W Cooper

```
        return pv;
    }
    //-----
    public void addArgs(MultVar mv) {
        args = mv.getVector ();
    }
```

Reducing the Parsed Stack

The tokens on the stack have this form.

```
Var
Var
Verb
Var
Var
Var
Var
Verb
```

We reduce the stack a token at a time, folding successive Vars into a MultVar class until the arguments are folded into the verb objects, as we show in Figure 23-2

Copyright © , 2002 by James W Cooper

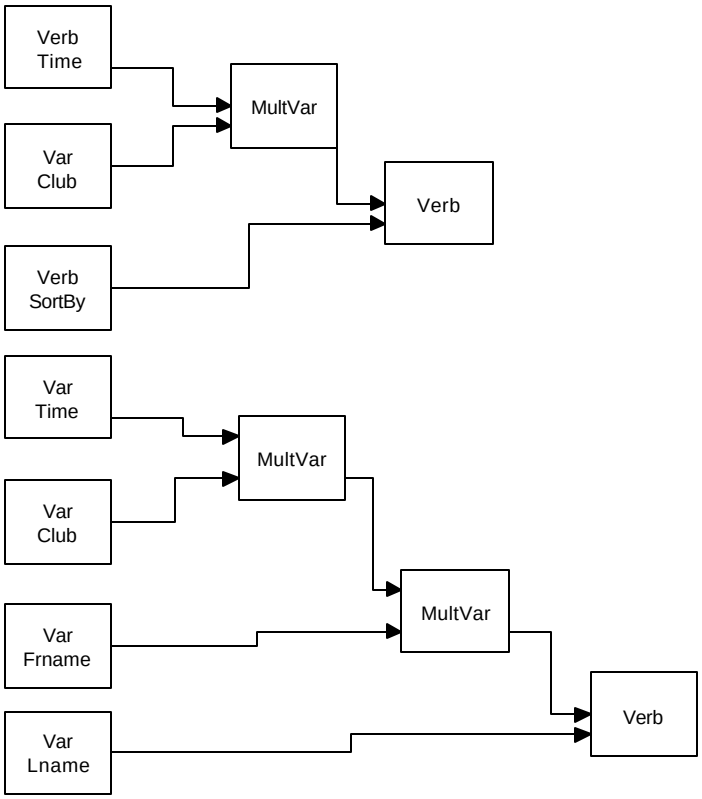


Figure 23-2– How the stack is reduced during parsing

When the stack reduces to a verb, this verb and its arguments are placed in an action list; when the stack is empty, the actions are executed.

Creating a Parser class that is a Command object and executing it when the Go button is pressed on the user interface carries out this entire process.

```
private void btCompute_Click(object sender, EventArgs e) {
       parse();
}
private void parse() {
       Parser par = new Parser (txCommand.Text ,kdata, lsResults);
```

Copyright © , 2002 by James W Cooper

```
        par.Execute ();
}
```

The parser itself just reduces the tokens, as the preceding shows. It checks for various pairs of tokens on the stack and reduces each pair to a single one for each of five different cases.

Implementing the Interpreter Pattern

It would certainly be possible to write a parser for this simple grammar as just a series of *if* statements. For each of the six possible stack configurations, reduce the stack until only a verb remains. Then, since we have made the Print and Sort verb classes Command objects, we can just Execute them one by one as the action list is enumerated.

However, the real advantage of the Interpreter pattern is its flexibility. By making each parsing case an individual object, we can represent the parse tree as a series of connected objects that reduce the stack successively. Using this arrangement, we can easily change the parsing rules without much in the way of program changes: We just create new objects and insert them into the parse tree.

According to the Gang of Four, these are the names for the participating objects in the Interpreter pattern.:

- **AbstractExpression**—declares the abstract Interpret operation.
- **TerminalExpression**—interprets expressions containing any of the terminal tokens in the grammar.
- **NonTerminalExpression**—interprets all of the nonterminal expressions in the grammar.
- **Context**—contains the global information that is part of the parser—in this case, the token stack.
- **Client**—Builds the syntax tree from the preceding expression types and invokes the Interpret operation.

Copyright © , 2002 by James W Cooper

The Syntax Tree

The syntax tree we construct to carry out the parsing of the stack we just showed can be quite simple. We just need to look for each of the stack configurations we defined and reduce them to an executable form. In fact, the best way to implement this tree is using a Chain of Responsibility, which passes the stack configuration along between classes until one of them recognizes that configuration and acts on it. You can decide whether a successful stack reduction should end that pass or not. It is perfectly possible to have several successive chain members work on the stack in a single pass. The processing ends when the stack is empty. We see a diagram of the individual parse chain elements in Figure 23-3.

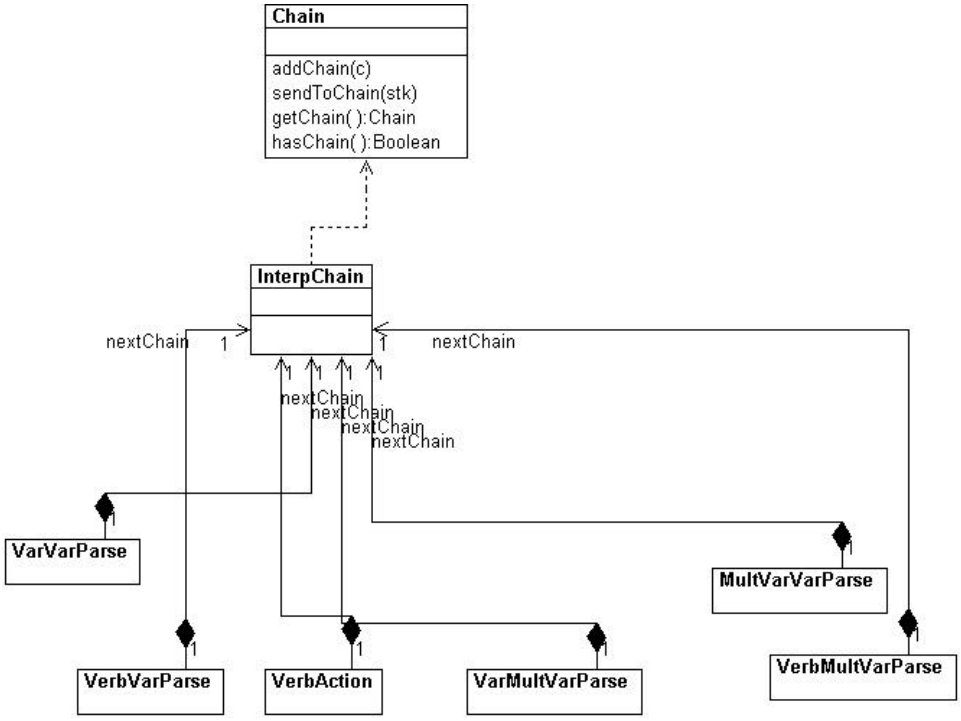


Figure 23-3– How the classes that perform the parsing interact.

Copyright © , 2002 by James W Cooper

In this class structure, we start with the AbstractExpression interpreter class *InterpChain.*

```
public abstract class InterpChain:Chain {
       private Chain nextChain;
       protected Stack stk;
       private bool hasChain;
       //-----
       public InterpChain()       {
              stk = new Stack ();
              hasChain = false;
       }
       //-----
       public void addToChain(Chain c) {
              nextChain = c;
              hasChain = true;
       }
       //-----
       public abstract bool interpret();
       //-----
       public void sendToChain(Stack stack) {
              stk = stack;
              if(! interpret()  ) {            //interpret stack
                     nextChain.sendToChain (stk);     //pass along
              }
       }
       //-----
       public bool topStack(int c1, int c2) {
              ParseObject p1, p2;
              p1 = stk.top ();
              p2 = stk.nextTop ();
              try{
                return (p1.getType() == c1 && p2.getType() == c2);
              }
              catch(NullReferenceException) {
                     return false;
              }
       }
       //-----
       public void addArgsToVerb() {
              ParseObject p = (ParseObject) stk.pop();
              ParseVerb v =  (ParseVerb) stk.pop();
              v.addArgs (p);
              stk.push (v);
       }
```

Copyright © , 2002 by James W Cooper

```
        //-----
        public Chain getChain() {
                return nextChain;
        }
```

This class also contains the methods for manipulating objects on the stack. Each of the subclasses implements the *interpret* operation differently and reduces the stack accordingly. For example, the complete VarVarParse class reduces two variables on the stack in succession to a single MultVar object.

```
public class VarVarParse : InterpChain  {
        public override bool interpret() {
                if(topStack(ParseVar.VAR , ParseVar.VAR )) {
                        //reduces VAR VAR to MULTVAR
                        ParseVar v1 = (ParseVar) stk.pop();
                        ParseVar v2 = (ParseVar) stk.pop();
                        MultVar mv = new MultVar (v2, v1);
                        stk.push (mv);
                        return true;
                }
                else
                        return false;
        }
}
```

Thus, in this implementation of the pattern, the stack constitutes the Context participant. Each of the first five subclasses of *InterpChain* are NonTerminalExpression participants, and the ActionVerb class that moves the completed verb and action objects to the actionList constitutes the TerminalExpression participant.

The client object is the Parser class that builds the stack object list from the typed-in command text and constructs the Chain of Responsibility from the various interpreter classes. We showed most of the Parser class above already. However, it also implements the Command pattern and sends the stack through the chain until it is empty and then executes the verbs that have accumulated in the action list when its Execute method is called.

Copyright © , 2002 by James W Cooper

```
  //executes parse and interpretation of command line
public void Execute() {
       while(stk.hasMoreElements () ) {
              chn.sendToChain (stk);
       }
       //now execute the verbs
       for(int i=0; i< actionList.Count ; i++ ) {
              Verb v = (Verb)actionList[i];
              v.setData (dat, ptable);
              v.Execute ();
       }
}
```

The final visual program is shown in Figure 23-4.

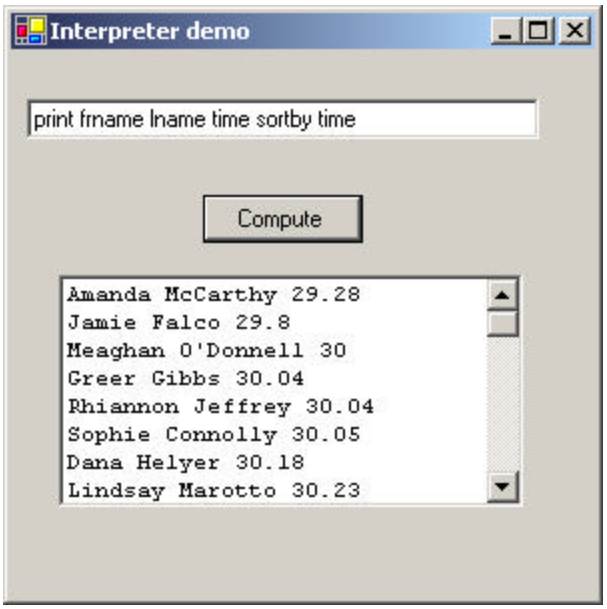


Figure 23-4 – The Interpreter pattern operating on the simple command in the text field

Copyright © , 2002 by James W Cooper

Consequences of the Interpreter Pattern

Whenever you introduce an interpreter into a program, you need to provide a simple way for the program user to enter commands in that language. It can be as simple as the Macro record button we noted earlier, or it can be an editable text field like the one in the preceding program.

However, introducing a language and its accompanying grammar also requires fairly extensive error checking for misspelled terms or misplaced grammatical elements. This can easily consume a great deal of programming effort unless some template code is available for implementing this checking. Further, effective methods for notifying the users of these errors are not easy to design and implement.

In the preceding Interpreter example, the only error handling is that keywords that are not recognized are not converted to ParseObjects and pushed onto the stack. Thus, nothing will happen because the resulting stack sequence probably cannot be parsed successfully, or if it can, the item represented by the misspelled keyword will not be included.

You can also consider generating a language automatically from a user interface of radio and command buttons and list boxes. While it may seem that having such an interface obviates the necessity for a language at all, the same requirements of sequence and computation still apply. When you have to have a way to specify the order of sequential operations, a language is a good way to do so, even if the language is generated from the user interface.

The Interpreter pattern has the advantage that you can extend or revise the grammar fairly easily once you have built the general parsing and reduction tools. You can also add new verbs or variables easily once the foundation is constructed. However, as the syntax of the grammar becomes more complex, you run the risk of creating a hard-to-maintain program.

Copyright © , 2002 by James W Cooper

While interpreters are not all that common in solving general programming problems, the Iterator pattern we take up next is one of the most common ones you'll be using.

Thought Question

Design a system to compute the results of simple quadratic expressions such as

```
4x^2 + 3x –4
```

where the user can enter x or a range of x's and can type in the equation.

Programs on the CD-ROM

`\Interpreter`	C# interpreter

Copyright © , 2002 by James W Cooper

24. The Iterator Pattern

The Iterator is one of the simplest and most frequently used of the design patterns. The Iterator pattern allows you to move through a list or collection of data using a standard interface without having to know the details of the internal representations of that data. In addition, you can also define special iterators that perform some special processing and return only specified elements of the data collection.

Motivation

The Iterator is useful because it provides a defined way to move through a set of data elements without exposing what is taking place inside the class. Since the Iterator is an *interface,* you can implement it in any way that is convenient for the data you are returning. *Design Patterns* suggests that a suitable interface for an Iterator might be the following.

```
public interface Iterator  {
       object First();
       object Next();
       bool isDone();
       object currentItem();
}
```

Here you can move to the top of the list, move through the list, find out if there are more elements, and find the current list item. This interface is easy to implement and it has certain advantages, but a number of other similar interfaces are possible. For example, when we discussed the Composite pattern, we introduced the getSubordinates method

```
IEnumerator getSubordinates();      //get subordinates
```

to provide a way to loop through all of the subordinates any employee may have. The IEnumerator interface can be represented in C# as

```
bool MoveNext();
void Reset();
object Current {get;}
```

Copyright © , 2002 by James W Cooper

This also allows us to loop through a list of zero or more elements in some internal list structure without our having to know how that list is organized inside the class.

One disadvantage of this Enumeration over similar constructs in C++ and Smalltalk is the strong typing of the C# language. This prevents the *Current()* property from returning an object of the actual type of the data in the collection. Instead, you must convert the returned object type to the actual type of the data in the collection. Thus, while this IEnumerator interface is intended to be polymorphic, this is not directly possible in C#.

Sample Iterator Code

Let's reuse the list of swimmers, clubs, and times we described earlier, and add some enumeration capabilities to the KidData class. This class is essentially a collection of Kids, each with a name, club, and time, and these Kid objects are stored in an ArrayList.

```
public class KidData :IEnumerator{
       private ArrayList kids;
       private int index;
       public KidData(string filename)         {
              kids = new ArrayList ();
              csFile fl = new csFile (filename);
              fl.OpenForRead ();
              string line = fl.readLine ();
              while(line != null) {
                     Kid kd = new Kid (line);
                     kids.Add (kd);
                     line = fl.readLine ();
              }
              fl.close ();
              index = 0;
       }
```

To obtain an enumeration of all the Kids in the collection, we simply use the methods of the IEnumerator interface we just defined.

```
public bool MoveNext() {
       index++;
       return index < kids.Count ;
}
```

Copyright © , 2002 by James W Cooper

```
//------
public object Current {
       get {
                return kids[index];
          }
}
//------
public void Reset() {
       index = 0;
}
```

Reading in the data and displaying a list of names is quite easy. We initialize the Kids class with the filename and have it build the collection of kid objects. Then we treat the Kids class as an instance of IEnumerator and move through it to get out the kids and display their names.

```
private void init() {
  kids = new KidData("50free.txt");
  while (kids.MoveNext () ) {
       Kid kd = (Kid)kids.Current ;
       lsKids.Items.Add (kd.getFrname()+ " "+ kd.getLname ());
  }
}
```

Fetching an Iterator

Another slightly more flexible way to handle iterators in a class is to provide the class with a getIterator method that returns instances of an iterator for that class's data. This is somewhat more flexible because you can have any number of iterators active simultaneously on the same data. Our KidIterator class can then be the one that implements our Iterator interface.

```
public class KidIterator : IEnumerator  {
       private ArrayList kids;
       private int index;
       public KidIterator(ArrayList kidz)      {
              kids = kidz;
              index = 0;
       }
       //------
       public bool MoveNext() {
```

Copyright © , 2002 by James W Cooper

```
            index++;
            return index < kids.Count ;
        }
        //------
        public object Current {
            get {
                return kids[index];
            }
        }
        //------
        public void Reset() {
            index = 0;
        }
}
```

We can fetch iterators from the main KidList class by creating them as needed.

```
public KidIterator getIterator() {
        return new KidIterator (kids);
}
```

Filtered Iterators

While having a clearly defined method of moving through a collection is helpful, you can also define filtered Iterators that perform some computation on the data before returning it. For example, you could return the data ordered in some particular way or only those objects that match a particular criterion. Then, rather than have a lot of very similar interfaces for these filtered iterators, you simply provide a method that returns each type of enumeration with each one of these enumerations having the same methods.

The Filtered Iterator

Suppose, however, that we wanted to enumerate only those kids who belonged to a certain club. This necessitates a special Iterator class that has access to the data in the KidData class. This is very simple because the methods we just defined give us that access. Then we only need to write an Iterator that only returns kids belonging to a specified club.

Copyright © , 2002 by James W Cooper

```
public class FilteredIterator : IEnumerator    {
       private ArrayList kids;
       private int index;
       private string club;
       public FilteredIterator(ArrayList kidz, string club) {
              kids = kidz;
              index = 0;
              this.club = club;
       }
       //------
       public bool MoveNext() {
              bool more = index < kids.Count-1 ;
              if(more) {
                     Kid kd = (Kid)kids[++index];
                     more = index < kids.Count;
                     while(more && ! kd.getClub().Equals (club)) {
                            kd = (Kid)kids[index++];
                            more = index < kids.Count ;
                     }
              }
              return more;
       }
       //------
       public object Current {
              get {
                     return kids[index];
              }
       }
       //------
       public void Reset() {
              index = 0;
       }
}
```

All of the work is done in the *MoveNext()* method, which scans through the collection for another kid belonging to the club specified in the constructor. Then it returns either true or false.

Finally, we need to add a method to KidData to return this new filtered Enumeration.

```
public FilteredIterator getFilteredIterator(string club) {
       return new FilteredIterator (kids, club);
```

Copyright © , 2002 by James W Cooper

```
}
```

This simple method passes the collection to the new Iterator class FilteredIterator along with the club initials. A simple program is shown in Figure 24-1 that displays all of the kids on the left side. It fills a combo box with a list of the clubs and then allows the user to select a club and fills the right-hand list box with those belonging to a single club. The class diagram is shown in Figure 24-2. Note that the *elements* method in KidData supplies an Enumeration and the kidClub class is in fact itself an Enumeration class.

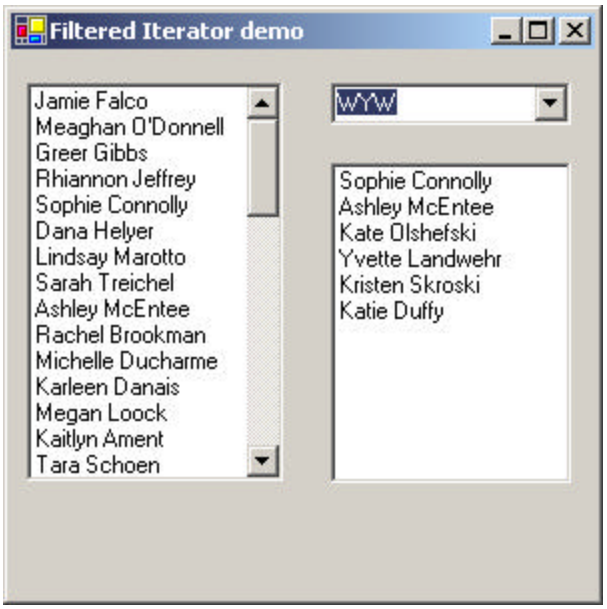


Figure 24-1 – A simple program-illustrated filtered enumeration

Copyright © , 2002 by James W Cooper

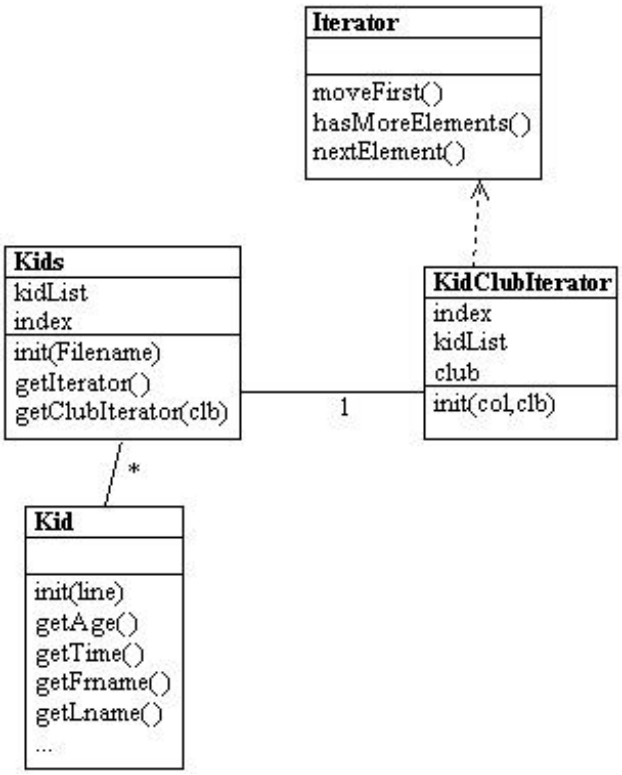


Figure 24-2– The classes used in the Filtered enumeration

Keeping Track of the Clubs

We need to obtain a unique list of the clubs to load the combo box in Figure 25-1 with. As we read in each kid, we can do this by putting the clubs in a Hashtable:

```
while(line != null) {
       Kid kd = new Kid (line);
       string club = kd.getClub ();
       if(! clubs.Contains (club ) ) {
              clubs.Add (club, club);
       }
```

Copyright © , 2002 by James W Cooper

```
        kids.Add (kd);
        line = fl.readLine ();
}
```

Then when we want to get the list of clubs, we can ask the Hashtable for an iterator of its contents. The Hashtable class has a method getEnumerator which should return this information. However, this method returns an IdictionaryEnumerator, which is slightly different. While it is derived from IEnumerator, it uses a Value method to return the contents of the hash table. This, we load the combo box with the following code:

```
IDictionaryEnumerator clubiter = kdata.getClubs ();
while(clubiter.MoveNext ()) {
        cbClubs.Items.Add ((string)clubiter.Value );
}
```

When we click on the combo box, it gets the selected club to generate a filtered iterator and load the kidclub list box:

```
private void cbClubs_SelectedIndexChanged(object sender,
                            EventArgs e) {
        string club = (String)cbClubs.SelectedItem ;
        FilteredIterator iter = kdata.getFilteredIterator ( club);
        lsClubKids.Items.Clear ();
        while(iter.MoveNext() ) {
          Kid kd = (Kid) iter.Current;
          lsClubKids.Items.Add (kd.getFrname() +" "+
                     kd.getLname ());
        }
}
```

Consequences of the Iterator Pattern

1. *Data modification.* The most significant question iterators may raise is the question of iterating through data while it is being changed. If your code is wide ranging and only occasionally moves to the next element, it is possible that an element might be added or deleted from the underlying collection while you are moving through it. It is also

Copyright © , 2002 by James W Cooper

possible that another thread could change the collection. There are no simple answers to this problem. If you want to move through a loop using an Enumeration and delete certain items, you must be careful of the consequences. Deleting or adding an element might mean that a particular element is skipped or accessed twice, depending on the storage mechanism you are using.

2. *Privileged access.* Enumeration classes may need to have some sort of privileged access to the underlying data structures of the original container class so they can move through the data. If the data is stored in an Arraylist or Hashtable, this is pretty easy to accomplish, but if it is in some other collection structure contained in a class, you probably have to make that structure available through a *get* operation. Alternatively, you could make the Iterator a derived class of the containment class and access the data directly.
3. *External versus Internal Iterators.* The *Design Patterns* text describes two types of iterators: external and internal. Thus far, we have only described external iterators. Internal iterators are methods that move through the entire collection, performing some operation on each element directly without any specific requests from the user. These are less common in C#, but you could imagine methods that normalized a collection of data values to lie between 0 and 1 or converted all of the strings to a particular case. In general, external iterators give you more control because the calling program accesses each element directly and can decide whether to perform an operation on it.

Programs on the CD-ROM

`\Iterator\SimpleIterator`	kid list using Iterator
`\Iterator\FilteredIterator`	filtered iterator by team name

Copyright © , 2002 by James W Cooper

25. The Mediator Pattern

When a program is made up of a number of classes, the logic and computation is divided logically among these classes. However, as more of these isolated classes are developed in a program, the problem of communication between these classes become more complex. The more each class needs to know about the methods of another class, the more tangled the class structure can become. This makes the program harder to read and harder to maintain. Further, it can become difficult to change the program, since any change may affect code in several other classes. The Mediator pattern addresses this problem by promoting looser coupling between these classes. Mediators accomplish this by being the only class that has detailed knowledge of the methods of other classes. Classes inform the Mediator when changes occur, and the Mediator passes on the changes to any other classes that need to be informed.

An Example System

Let's consider a program that has several buttons, two list boxes, and a text entry field, as shown in Figure 25-1.

Copyright © , 2002 by James W Cooper

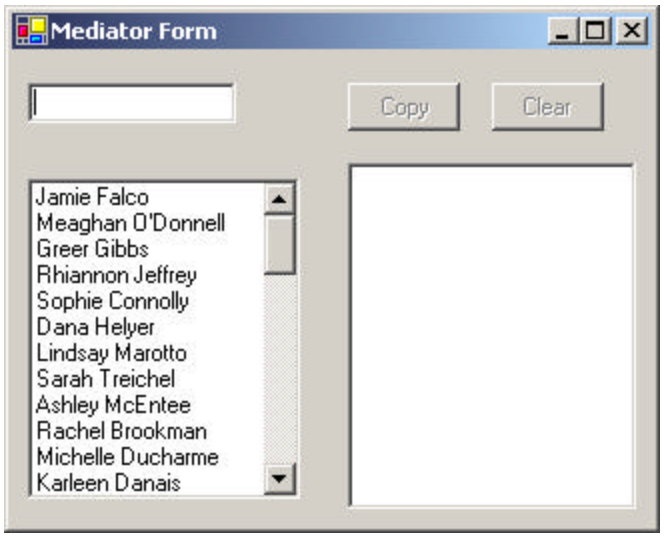


Figure 25-1– A simple program with two lists, two buttons, and a text field that will interact

When the program starts, the Copy and Clear buttons are disabled.

1. When you select one of the names in the left-hand list box, it is copied into the text field for editing, and the *Copy* button is enabled.
2. When you click on *Copy*, that text is added to the right-hand list box, and the *Clear* button is enabled, as we see in Figure 25-2.

Copyright © , 2002 by James W Cooper

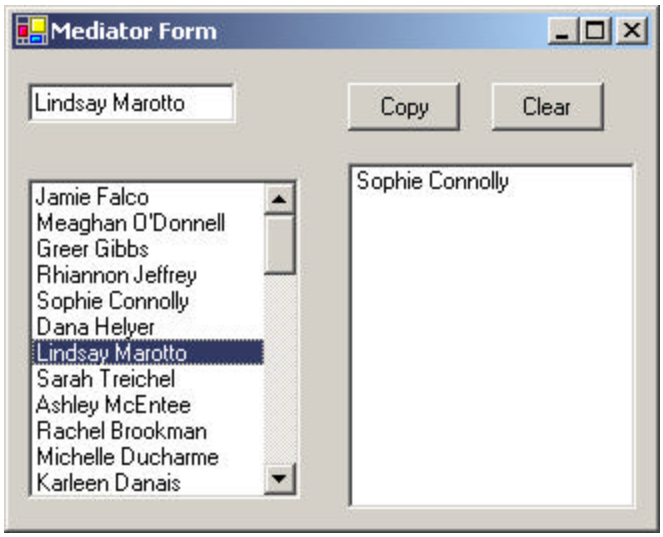


Figure 25-2 – When you select a name, the buttons are enabled, and when you click on Copy, the name is copied to the right list box.

3. If you click on the *Clear* button, the right-hand list box and the text field are cleared, the list box is deselected, and the two buttons are again disabled.

User interfaces such as this one are commonly used to select lists of people or products from longer lists. Further, they are usually even more complicated than this one, involving insert, delete, and undo operations as well.

Interactions Between Controls

The interactions between the visual controls are pretty complex, even in this simple example. Each visual object needs to know about two or more others, leading to quite a tangled relationship diagram, as shown in Figure 25-3.

Copyright © , 2002 by James W Cooper

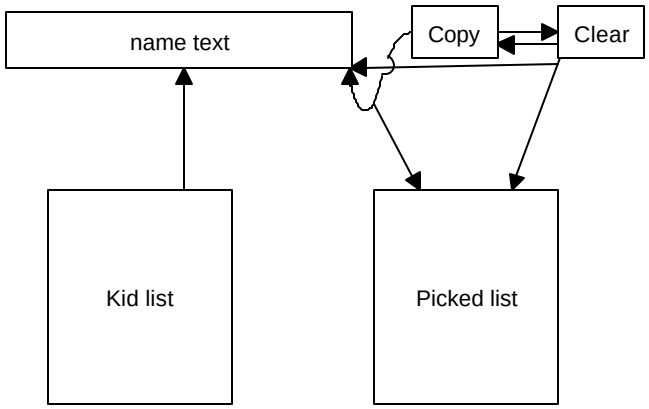


Figure 25-3 – A tangled web of interactions between classes in the simple visual interface we presented in and Figure 25-1 and Figure 25-2.

The Mediator pattern simplifies this system by being the only class that is aware of the other classes in the system. Each of the controls with which the Mediator communicates is called a Colleague. Each Colleague informs the Mediator when it has received a user event, and the Mediator decides which other classes should be informed of this event. This simpler interaction scheme is illustrated in Figure 25-4.

Copyright © , 2002 by James W Cooper

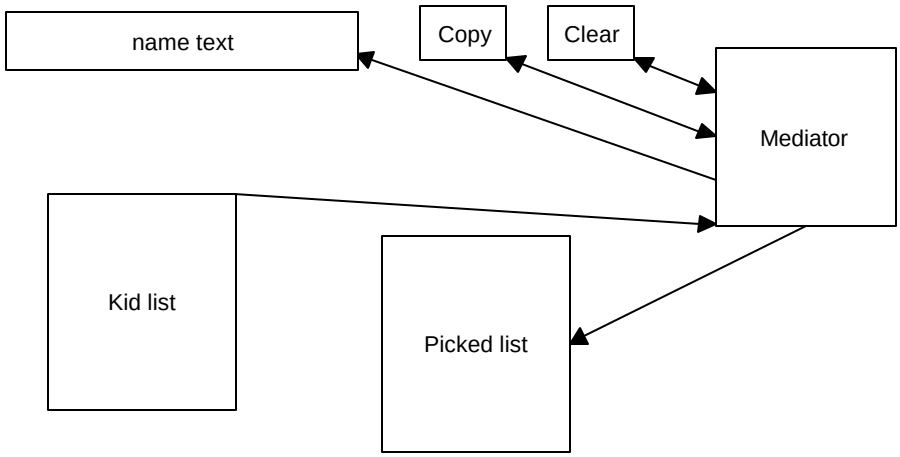


Figure 25-4 – A Mediator class simplifies the interactions between classes.

The advantage of the Mediator is clear: It is the only class that knows of the other classes and thus the only one that would need to be changed if one of the other classes changes or if other interface control classes are added.

Sample Code

Let's consider this program in detail and decide how each control is constructed. The main difference in writing a program using a Mediator class is that each class needs to be aware of the existence of the Mediator. You start by creating an instance of your Mediator class and then pass the instance of the Mediator to each class in its constructor.

```
med = new Mediator (btCopy, btClear, lsKids, lsSelected);
btCopy.setMediator (med);  //set mediator ref in each control
btClear.setMediator (med);
lsKids.setMediator (med);
med.setText (txName);          //tell mediator about text
box
```

Copyright © , 2002 by James W Cooper

We derive our two button classes from the Button class, so they can also implement the Command interface. These buttons are passed to the Mediator in its constructor. Here is the CpyButton class.

```
public class CpyButton : System.Windows.Forms.Button, Command {
       private Container components = null;
       private Mediator med;
       //-----
       public CpyButton()         {
              InitializeComponent();
       }
       //-----
       public void setMediator(Mediator md) {
              med = md;
       }
       //-----
       public void Execute() {
              med.copyClicked ();
       }
```

It's Execute method simply tells the Mediator lass that it has been clicked, and lets the Mediator decide what to do when this happens. The Clear button is exactly analogous.

We derive the KidList class from the ListBox class and have it loaded with names within the Mediator's constructor.

```
public Mediator(CpyButton cp, ClrButton clr, KidList kl,
                           ListBox pk)          {
       cpButton = cp;                      //copy in buttons
       clrButton = clr;
       klist = kl;                         //copy in list boxes
       pkList = pk;
       kds = new KidData ("50free.txt"); //create data list class
       clearClicked();                     //clear all controls
       KidIterator kiter = kds.getIterator ();
       while(kiter.MoveNext () ) {             //load list box
              Kid kd = (Kid) kiter.Current ;
              klist.Items .Add (kd.getFrname() +" "+
                           kd.getLname ());
       }
}
```

Copyright © , 2002 by James W Cooper

We don't have to do anything special to the text field, since all its activity takes place within the Mediator; we just pass it to the Mediator using as setText method as we illustrated above.

The only other important part of our initialization is creating a single event handler for the two buttons and the list box. Rather than letting the development environment generate these click events for us, we create a single event and add it to the click handlers for the two buttons and the list box's SelectIndexChanged event. The intriguing thing about this event handler is that all it needs to do is call each control's Execute method and let the Mediator methods called by those Execute methods do all the real work.

The event handler for these click events is simply

```
//each control is a command object
public void clickHandler(object obj, EventArgs e) {
       Command comd = (Command)obj;      //get command object
       comd.Execute ();                  //and execute command
}
```

We show the complete Form initialization method that creates this event connections below:

```
private void init() {
       //set up mediator and pass in referencs to controls
       med = new Mediator (btCopy, btClear, lsKids, lsSelected);
       btCopy.setMediator (med);  // mediator ref in each control
       btClear.setMediator (med);
       lsKids.setMediator (med);
       med.setText (txName);      //tell mediator about text box

       //create event handler for all command objects
       EventHandler evh = new EventHandler (clickHandler);
       btClear.Click += evh;
       btCopy.Click += evh;
       lsKids.SelectedIndexChanged += evh;
}
```

Copyright © , 2002 by James W Cooper

The general point of all these classes is that each knows about the Mediator and tells the Mediator of its existence so the Mediator can send commands to it when appropriate.

The Mediator itself is very simple. It supports the Copy, Clear, and Select methods and has a register method for the TextBox. The two buttons and the ListBox are passed in in the Mediator's constructor. Note that there is no real reason to choose setXxx methods over constructor arguments for passing in references to these controls. We simple illustrate both approaches in this example.

```
public class Mediator     {
       private CpyButton cpButton;      //buttons
       private ClrButton clrButton;
       private TextBox txKids;          //text box
       private ListBox pkList;          //list boxes
       private KidList klist;
       private KidData kds;             //list of data from file

       public Mediator(CpyButton cp, ClrButton clr,
              KidList kl, ListBox pk)          {
              cpButton = cp;            //copy in buttons
              clrButton = clr;
              klist = kl;               //copy in list boxes
              pkList = pk;
              kds = new KidData ("50free.txt"); //create data list
              clearClicked();           //clear all controls
              KidIterator kiter = kds.getIterator ();
              while(kiter.MoveNext () ) {      //load list box
                     Kid kd = (Kid) kiter.Current ;
                     klist.Items .Add (kd.getFrname() +
                            " "+kd.getLname ());
              }
       }
       //-----
       //get text box reference
              public void setText(TextBox tx) {
                     txKids = tx;
              }
       //-----
       //clear lists and set buttons to disabled
       public void clearClicked() {
              //disable buttons and clear list
```

Copyright © , 2002 by James W Cooper

```
            cpButton.Enabled = false;
            clrButton.Enabled = false;
            pkList.Items.Clear();
        }
        //-----
        //copy data from text box to list box
        public void copyClicked() {
             //copy name to picked list
            pkList.Items.Add(txKids.Text);
            //clear button enabled
            clrButton.Enabled = true;
            klist.SelectedIndex = -1;
        }
        //-----
        //copy selected kid to text box
        //enable copy button
        public void kidPicked() {
            //copy text from list to textbox
            txK    ids.Text = klist.Text;
            //copy button enabled
            cpButton.Enabled = true;
        }
}
```

Initialization of the System

One further operation that is best delegated to the Mediator is the initialization of all the controls to the desired state. When we launch the program, each control must be in a known, default state, and since these states may change as the program evolves, we simply carry out this initialization in the Mediator's constructor, which sets all the controls to the desired state. In this case, that state is the same as the one achieved by the Clear button, and we simply call that method this.

```
        clearClicked();            //clear all controls
```

Mediators and Command Objects

The two buttons in this program use command objects. Just as we noted earlier, this makes processing of the button click events quite simple.

Copyright © , 2002 by James W Cooper

In either case, however, this represents the solution to one of the problems we noted in the Command pattern chapter: Each button needed knowledge of many of the other user interface classes in order to execute its command. Here, we delegate that knowledge to the Mediator, so the Command buttons do not need any knowledge of the methods of the other visual objects. The class diagram for this program is shown in Figure 25-5, illustrating both the Mediator pattern and the use of the Command pattern.

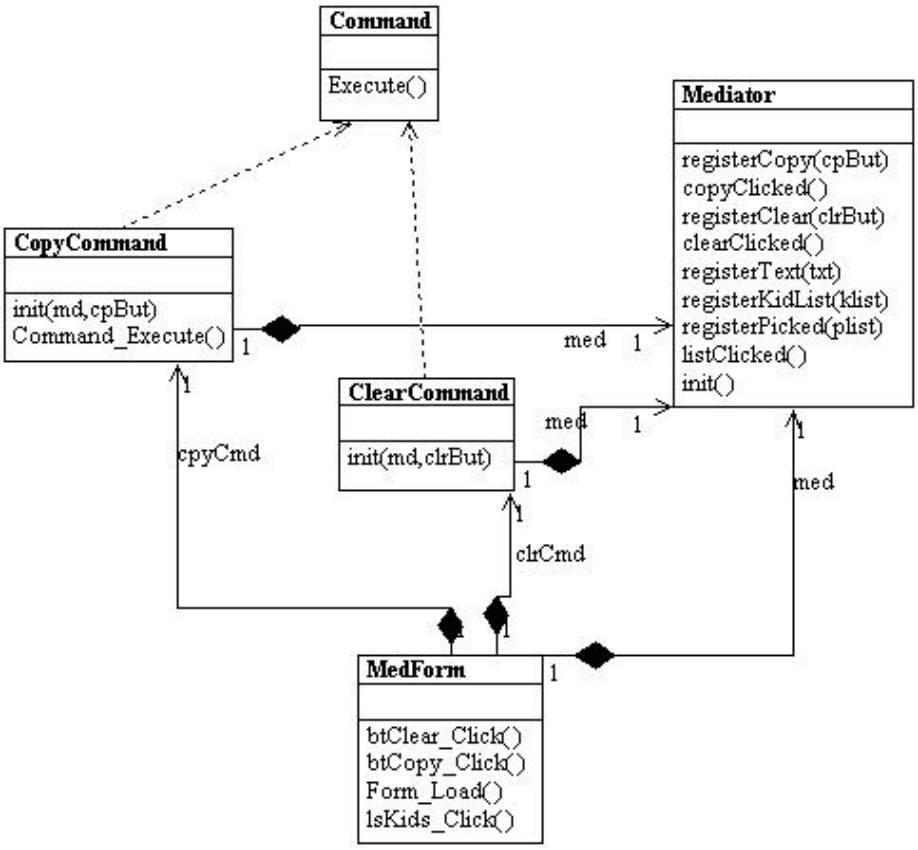


Figure 25-5 – The interactions between the Command objects and the Mediator object

Copyright © , 2002 by James W Cooper

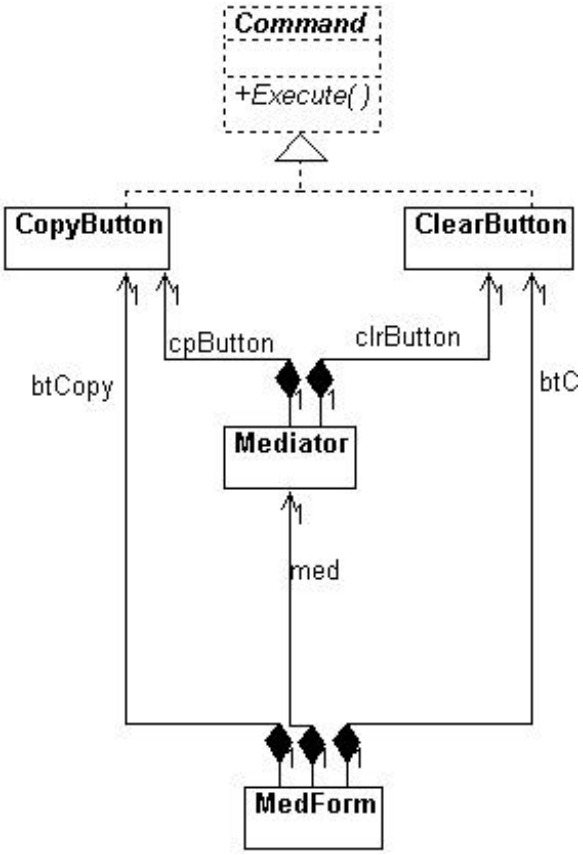


Figure 25-6 – The UML diagram for the C# Mediator pattern

Consequences of the Mediator Pattern

1. The Mediator pattern keeps classes from becoming entangled when actions in one class need to be reflected in the state of another class.
2. Using a Mediator makes it easy to change a program's behavior. For many kinds of changes, you can merely change or subclass the Mediator, leaving the rest of the program unchanged.

Copyright © , 2002 by James W Cooper

3. You can add new controls or other classes without changing anything except the Mediator.
4. The Mediator solves the problem of each Command object needing to know too much about the objects and methods in the rest of a user interface.
5. The Mediator can become a "god class," having too much knowledge of the rest of the program. This can make it hard to change and maintain. Sometimes you can improve this situation by putting more of the function into the individual classes and less into the Mediator. Each object should carry out its own tasks, and the Mediator should only manage the interaction between objects.
6. Each Mediator is a custom-written class that has methods for each Colleague to call and knows what methods each Colleague has available. This makes it difficult to reuse Mediator code in different projects. On the other hand, most Mediators are quite simple, and writing this code is far easier than managing the complex object interactions any other way.

Single Interface Mediators

The Mediator pattern described here acts as a kind of Observer pattern, observing changes in each of the Colleague elements, with each element having a custom interface to the Mediator. Another approach is to have a single interface to your Mediator and pass to that method various objects that tell the Mediator which operations to perform.

In this approach, we avoid registering the active components and create a single action method with different polymorphic arguments for each of the action elements.

```
public void action(MoveButton mv);
public void action(clrButton clr);
public void action(KidList klist);
```

Copyright © , 2002 by James W Cooper

Thus, we need not register the action objects, such as the buttons and source list boxes, since we can pass them as part of generic *action* methods.

In the same fashion, you can have a single Colleague interface that each Colleague implements, and each Colleague then decides what operation it is to carry out.

Implementation Issues

Mediators are not limited to use in visual interface programs; however, it is their most common application. You can use them whenever you are faced with the problem of complex intercommunication between a number of objects.

Programs on the CD-ROM

`\Mediator`	Mediator

Copyright © , 2002 by James W Cooper

26. The Memento Pattern

In this chapter, we discuss how to use the Memento pattern to save data about an object so you can restore it later. For example, you might like to save the color, size, pattern, or shape of objects in a drafting or painting program. Ideally, it should be possible to save and restore this state without making each object take care of this task and without violating encapsulation. This is the purpose of the Memento pattern.

Motivation

Objects normally shouldn't expose much of their internal state using public methods, but you would still like to be able to save the entire state of an object because you might need to restore it later. In some cases, you could obtain enough information from the public interfaces (such as the drawing position of graphical objects) to save and restore that data. In other cases, the color, shading, angle, and connection relationships to other graphical objects need to be saved, and this information is not readily available. This sort of information saving and restoration is common in systems that need to support Undo commands.

If all of the information describing an object is available in public variables, it is not that difficult to save them in some external store. However, making these data public makes the entire system vulnerable to change by external program code, when we usually expect data inside an object to be private and encapsulated from the outside world.

The Memento pattern attempts to solve this problem in some languages by having privileged access to the state of the object you want to save. Other objects have only a more restricted access to the object, thus preserving their encapsulation. In C#, however, there is only a limited notion of privileged access, but we will make use of it in this example.

This pattern defines three roles for objects.

1. The **Originator** is the object whose state we want to save.
2. The **Memento** is another object that saves the state of the Originator.

Copyright © , 2002 by James W Cooper

3. The **Caretaker** manages the timing of the saving of the state, saves the Memento, and, if needed, uses the Memento to restore the state of the Originator.

Implementation

Saving the state of an object without making all of its variables publicly available is tricky and can be done with varying degrees of success in various languages. *Design Patterns* suggests using the C++ *friend* construction to achieve this access, and the *Smalltalk Companion* notes that it is not directly possible in Smalltalk. In Java, this privileged access is possible using the package protected mode. The *internal* keyword is available in C#, but all that means is that any class method labeled as *internal* will only be accessible within the project. If you make a library from such classes, the methods marked as internal will not be exported and available. Instead, we will define a property to fetch and store the important internal values and make use of no other properties for any purpose in that class. For consistency, we'll use the internal keyword on these properties, but remember that this linguistic use of *internal* is not very restrictive.

Sample Code

Let's consider a simple prototype of a graphics drawing program that creates rectangles and allows you to select them and move them around by dragging them with the mouse. This program has a toolbar containing three buttons—Rectangle, Undo, and Clear—as we see in Figure 26-1

Copyright © , 2002 by James W Cooper

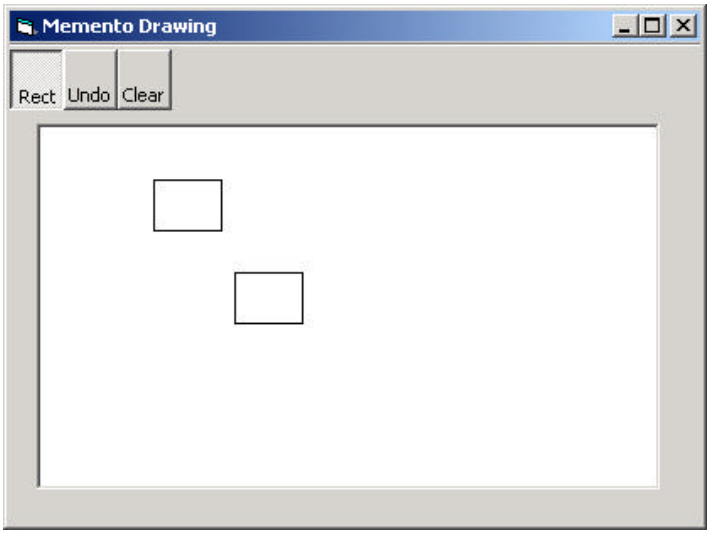


Figure 26-1 – A simple graphics drawing program that allows you to draw rectangles, undo their drawing, and clear the screen

The Rectangle button is a toolbar ToggleButton that stays selected until you click the mouse to draw a new rectangle. Once you have drawn the rectangle, you can click in any rectangle to select it, as we see in Figure 26-2.

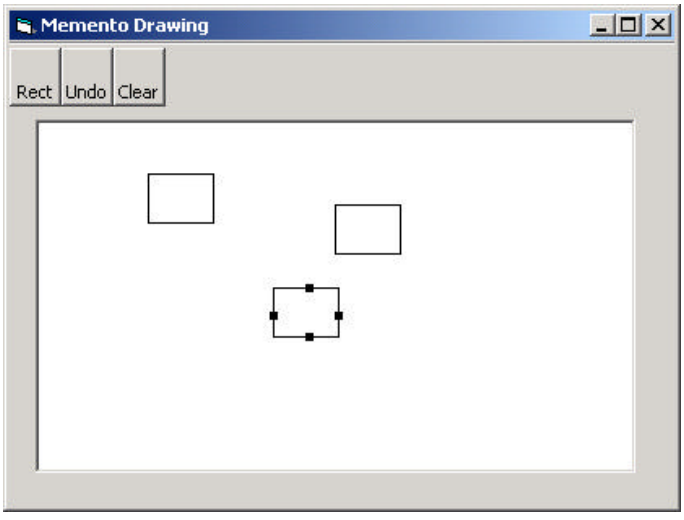


Figure 26-2– Selecting a rectangle causes "handles" to appear, indicating that it is selected and can be moved.

Copyright © , 2002 by James W Cooper

Once it is selected, you can drag that rectangle to a new position, using the mouse, as shown in Figure 26-3

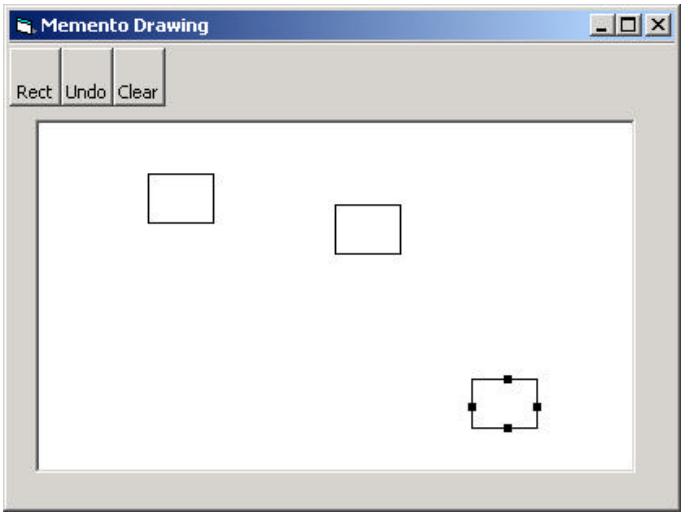


Figure 26-3 – The same selected rectangle after dragging

The Undo button can undo a succession of operations. Specifically, it can undo moving a rectangle, and it can undo the creation of each rectangle. There are five actions we need to respond to in this program.

1. Rectangle button click
2. Undo button click
3. Clear button click
4. Mouse click
5. Mouse drag

The three buttons can be constructed as Command objects, and the mouse click and drag can be treated as commands as well. Since we have a number of visual objects that control the display of screen objects, this suggests an opportunity to use the Mediator pattern, and that is, in fact, the way this program is constructed.

We will create a Caretaker class to manage the Undo action list. It can keep a list of the last *n* operations so they can be undone. The Mediator maintains the list of drawing objects and communicates with the

Copyright © , 2002 by James W Cooper

Caretaker object as well. In fact, since there could be any number of actions to save and undo in such a program, a Mediator is virtually required so there is a single place to send these commands to the Undo list in the Caretaker.

In this program, we save and undo only two actions: creating new rectangles and changing the position of rectangles. Let's start with our visRectangle class, which actually draws each instance of the rectangles.

```
public class VisRectangle       {
      private int x, y, w, h;
      private const int SIZE=30;
      private CsharpPats.Rectangle rect;
      private bool selected;
      private Pen bPen;
      private SolidBrush bBrush;
      //-----
      public VisRectangle(int xp, int yp)           {
            x = xp;                    y = yp;
            w = SIZE;                  h = SIZE;
            saveAsRect();
            bPen = new Pen(Color.Black);
            bBrush = new SolidBrush(Color.Black);
      }
      //-----
      //used by Memento for saving and restoring state
      internal CsharpPats.Rectangle rects {
            get {
                  return rect;
            }
            set {
                  x=value.x;
                  y=value.y;
                  w=value.w;
                  h=value.h;
                  saveAsRect();
            }
      }
      //------
      public void setSelected(bool b) {
            selected = b;
      }
      //-----
      //move to new position
      public void move(int xp, int yp) {
            x = xp;
            y = yp;
            saveAsRect();
```

Copyright © , 2002 by James W Cooper

```
}
//-----
public void draw(Graphics g) {
       //draw rectangle
       g.DrawRectangle(bPen, x, y, w, h);

  if (selected) {   //draw handles
    g.FillRectangle(bBrush, x + w / 2, y - 2, , 4);
    g.FillRectangle(bBrush, x - 2, y + h / 2, 4, 4);
    g.FillRectangle(bBrush, x + (w / 2), y + h - 2, 4, );
    g.FillRectangle(bBrush, x + (w - 2),
                            y + (h / 2), 4, 4);
  }
}
//-----
//return whether point is inside rectangle
public bool contains(int x, int y) {
       return rect.contains (x, y);
}
//------
//create Rectangle object from new position
private void saveAsRect() {
       rect = new CsharpPats.Rectangle (x,y,w,h);
}
```

We also use the same Rectangle class as we hace developed before, that contains Get and Set properties for the x, y, w, and h values and a *contains* method.

Drawing the rectangle is pretty straightforward. Now, let's look at our simple Memento class that we use to store the state of a rectangle.

```
public class Memento       {
       private int x, y, w, h;
       private CsharpPats.Rectangle rect;
       private VisRectangle visRect;
       //------
       public Memento(VisRectangle vrect)            {
              visRect = vrect;
              rect = visRect.rects ;
              x = rect.x ;
              y = rect.y;
              w = rect.w;
              h = rect.h;
       }
       //------
       public void restore() {
              rect.x = x;
              rect.y = y;
```

Copyright © , 2002 by James W Cooper

```
            rect.h = h;
            rect.w = w;
            visRect.rects = rect;
      }
}
```

When we create an instance of the Memento class, we pass it the visRectangle instance we want to save, using the init method. It copies the size and position parameters and saves a copy of the instance of the visRectangle itself. Later, when we want to restore these parameters, the Memento knows which instance to which it must restore them, and it can do it directly, as we see in the *restore()* method.

The rest of the activity takes place in the Mediator class, where we save the previous state of the list of drawings as an integer on the undo list.

```
public void createRect(int x, int y) {
      unpick();         //make sure none is selected
      if (startRect) {  //if rect button is depressed
            int count = drawings.Count;
            caretakr.Add(count);  //Save list size
            //create a rectangle
            VisRectangle v = new VisRectangle(x, y);
                  drawings.Add(v);//add element to list
            startRect = false;   //done with rectangle
            rect.setSelected(false);    //unclick button
            canvas.Refresh();
      }
      else
            //if not pressed look for rect to select
            pickRect(x, y);
      }
}
```

On the other hand, if you click on the panel when the Rectangle button has not been selected, you are trying to select an existing rectangle. This is tested here.

```
public void pickRect(int x, int y) {
      //save current selected rectangle
      //to avoid double save of undo
      int lastPick = -1;
      if (selectedIndex >= 0) {
            lastPick = selectedIndex;
      }
      unpick();  //undo any selection
```

Copyright © , 2002 by James W Cooper

```
        //see if one is being selected
        for (int i = 0; i< drawings.Count; i++) {
                VisRectangle v = (VisRectangle)drawings[i];
                if (v.contains(x, y)) {
                        //did click inside a rectangle
                        selectedIndex = i;     //save it
                        rectSelected = true;
                        if (selectedIndex != lastPick) {
                                //but don't save twice
                                caretakr.rememberPosition(v);
                        }
                        v.setSelected(true);    //turn on handles
                        repaint();          //and redraw
                }
        }
}
```

The Caretaker class remembers the previous position of the rectangle in a Memento object and adds it to the undo list.

```
public void rememberPosition(VisRectangle vr) {
        Memento mem = new Memento (vr);
        undoList.Add (mem);
}
```

The Caretaker class manages the undo list. This list is a Collection of integers and Memento objects. If the value is an integer, it represents the number of drawings to be drawn at that instant. If it is a Memento, it represents the previous state of a visRectangle that is to be restored. In other words, the undo list can undo the adding of new rectangles and the movement of existing rectangles.

Our undo method simply decides whether to reduce the drawing list by one or to invoke the *restore* method of a Memento. Since the undo list contains both integer objects and Memento objects, we cast the list element to a Memento type, and if this fails, we catch the cast exception and recognize that it will be a drawing list element to be removed.

```
public void undo() {
        if(undoList.Count > 0) {
                int last = undoList.Count -1;
                object obj = undoList[last];
                try{
                        Memento mem = (Memento)obj;
                        remove(mem);
                }
```

Copyright © , 2002 by James W Cooper

```
                catch (Exception) {
                        removeDrawing();
                }
                undoList.RemoveAt (last);
        }
}
```

The two remove methods either reduce the number of drawings or restore the position of a rectangle.

```
public void removeDrawing() {
        drawings.RemoveAt (drawings.Count -1);
}
public void remove(Memento mem) {
        mem.restore ();
}
```

A Cautionary Note

While it is helpful in this example to call out the differences between a Memento of a rectangle position and an integer specifying the addition of a new drawing, this is in general an absolutely terrible example of OO programming. You should *never* need to check the type of an object to decide what to do with it. Instead, you should be able to call the correct method on that object and have it do the right thing.

A more correct way to have written this example would be to have both the drawing element and the Memento class both have their own restore methods and have them both be members of a general Memento class (or interface). We take this approach in the State example pattern in the next chapter.

Command Objects in the User Interface

We can also use the Command pattern to help in simplifying the code in the user interface. You can build a toolbar and create ToolbarButtons in C# using the IDE, but if you do, it is difficult to subclass them to make them into command objects. There are two possible solutions. First, you can keep a parallel array of Command objects for the RectButton, the UndoButton, and the Clear button and call them in the toolbar click routine.

You should note, however, that the toolbar buttons do not have an Index property, and you cannot just ask which one has been clicked by

Copyright © , 2002 by James W Cooper

its index and relate it to the command array. Instead, we can use the GetHashCode property of each tool button to get a unique identifier for that button and keep the corresponding command objects in a Hashtable keyed off these button hash codes. We construct the Hashtable as follows.

```
private void init() {
       med = new Mediator(pic);     //create Mediator
       commands = new Hashtable();  //and Hash table
       //create the command objectsb
       RectButton rbutn = new RectButton(med, tbar.Buttons[0]);
       UndoButton ubutn = new UndoButton(med, tbar.Buttons[1]);
       ClrButton clrbutn = new ClrButton(med);
       med.registerRectButton (rbutn);
       //add them to the hashtable using the button hash values
       commands.Add(btRect.GetHashCode(), rbutn);
       commands.Add(btUndo.GetHashCode(), ubutn);
       commands.Add(btClear.GetHashCode(), clrbutn);
       pic.Paint += new PaintEventHandler (paintHandler);
}
```

Then the command interpretation devolves to just a few lines of code, since all the buttons call the same click event already. We can use these hash codes to get the right command object when the buttons are clicked.

```
private void tbar_ButtonClick(object sender,
                          ToolBarButtonClickEventArgs e) {
       ToolBarButton tbutn = e.Button ;
       Command comd = (Command)commands[tbutn.GetHashCode ()];
       comd.Execute ();
}
```

Alternatively, you could create the toolbar under IDE control but add the tool buttons to the collection programmatically and use derived buttons with a Command interface instead. We illustrate this approach in the State pattern.

The RectButton command class is where most of the activity takes place.

```
public class RectButton :  Command      {
       private ToolBarButton bt;
       private Mediator med;
       //------
       public RectButton(Mediator md, ToolBarButton tb)     {
              med = md;
              bt = tb;
       }
```

Copyright © , 2002 by James W Cooper

```
        //------
        public void setSelected(bool sel) {
                bt.Pushed  = sel;
        }
        //------
        public void Execute() {
                if(bt.Pushed  )
                        med.startRectangle ();
        }
}
```

Handling Mouse and Paint Events

We also must catch the mouse down, up, and move events and pass them on to the Mediator to handle.

```
private void pic_MouseDown(object sender, MouseEventArgs e) {
        mouse_down = true;
        med.createRect (e.X, e.Y);
}
//------
private void pic_MouseUp(object sender, MouseEventArgs e) {
        mouse_down = false;
}
//------
private void pic_MouseMove(object sender, MouseEventArgs e) {
        if(mouse_down)
                med.drag(e.X , e.Y);
}
```

Whenever the Mediator makes a change, it calls for a refresh of the picture box, which in turn calls the Paint event. We then pass this back to the Mediator to draw the rectangles in their new positions.

```
private void paintHandler(object sender, PaintEventArgs e ) {
        Graphics g =  e.Graphics ;
        med.reDraw (g);
}
```

The complete class structure is diagrammed in Figure 26-4

Copyright © , 2002 by James W Cooper

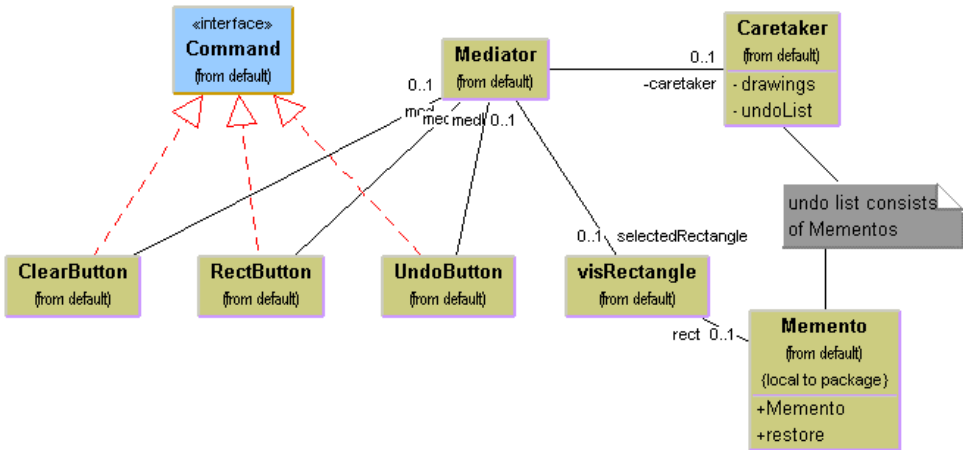


Figure 26-4 – The UML diagram for the drawing program using a Memento

Consequences of the Memento

The Memento provides a way to preserve the state of an object while preserving encapsulation in languages where this is possible. Thus, data to which only the Originator class should have access effectively remain private. It also preserves the simplicity of the Originator class by delegating the saving and restoring of information to the Memento class.

On the other hand, the amount of information that a Memento has to save might be quite large, thus taking up fair amounts of storage. This further has an effect on the Caretaker class that may have to design strategies to limit the number of objects for which it saves state. In our simple example, we impose no such limits. In cases where objects change in a predictable manner, each Memento may be able to get by with saving only incremental changes of an object's state.

In our example code in this chapter, we have to use not only the Memento but the Command and Mediator patterns as well. This clustering of several patterns is very common, and the more you see of good OO programs, the more you will see these pattern groupings.

Thought Question

Mementos can also be used to restore the state of an object when a process fails. If a database update fails because of a dropped network

Copyright © , 2002 by James W Cooper

connection, you should be able to restore the data in your cached data to their previous state. Rewrite the Database class in the Façade chapter to allow for such failures.

Programs on the CD-ROM

`\Memento`	Memento example

Copyright © , 2002 by James W Cooper

27. The Observer Pattern

In this chapter we discuss how you can use the Observer pattern to present data in several forms at once. In our new, more sophisticated windowing world, we often would like to display data in more than one form at the same time and have all of the displays reflect any changes in that data. For example, you might represent stock price changes both as a graph and as a table or list box. Each time the price changes, we'd expect both representations to change at once without any action on our part.

We expect this sort of behavior because there are any number of Windows applications, like Excel, where we see that behavior. Now there is nothing inherent in Windows to allow this activity, and, as you may know, programming directly in Windows in C or C++ is pretty complicated. In C#, however, we can easily use the Observer Design Pattern to make our program behave this way.

The Observer pattern assumes that the object containing the data is separate from the objects that display the data and that these display objects *observe* changes in that data. This is simple to illustrate, as we see in Figure 27-1.

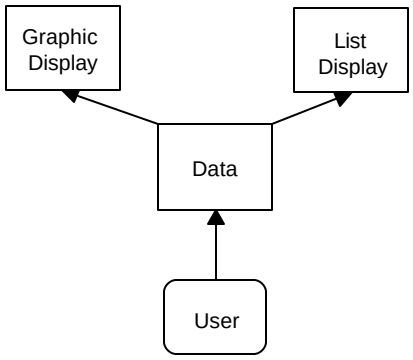


Figure 27-1– Data are displayed as a list and in some graphical mode.

Copyright © , 2002 by James W Cooper

When we implement the Observer pattern, we usually refer to the data as the Subject and each of the displays as an Observer. Each of these observers registers its interest in the data by calling a public method in the Subject. Then each observer has a known interface that the subject calls when the data change. We could define these interfaces as follows.

```
public interface Observer  {
      void sendNotify(string message);
//-----
public interface Subject   {
       void registerInterest(Observer obs);
}
```

The advantages of defining these abstract interfaces is that you can write any sort of class objects you want as long as they implement these interfaces and that you can declare these objects to be of type Subject and Observer no matter what else they do.

Watching Colors Change

Let's write a simple program to illustrate how we can use this powerful concept. Our program shows a display form containing three radio buttons named Red, Blue, and Green, as shown in Figure 27-2.

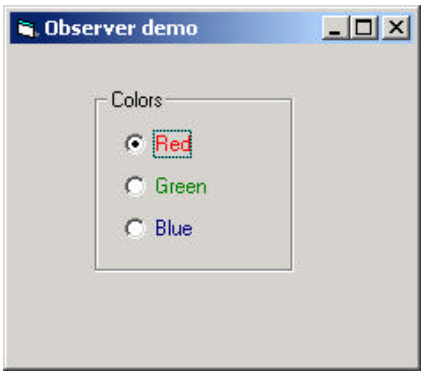


Figure 27-2 – A simple control panel to create red, green, or blue "data"

Now our main form class implements the Subject interface. That means that it must provide a public method for registering interest in the data in this class. This method is the *registerInterest* method, which just adds Observer objects to an ArrayList.

Copyright © , 2002 by James W Cooper

```
public void registerInterest(Observer obs ) {
       observers.Add (obs);
}
```

Now we create two observers, one that displays the color (and its name) and another that adds the current color to a list box. Each of these is actually a Windows form that also implements the Observer interface. When we create instances of these forms, we pass to them the base or startup form as an argument. Since this startup form is actually the Subject, they can register their interest in its events. So the main form's initialization creates these instances and passes them a reference to itself.

```
       ListObs lobs = new ListObs (this);
       lobs.Show ();
       ColObserver colObs = new ColObserver (this);
       colObs.Show();
```

Then, when we create our ListObs window, we register our interest in the data in the main program.

```
public ListObs(Subject subj)            {
       InitializeComponent();
       init(subj);
}
//------
public void init(Subject subj) {
       subj.registerInterest (this);
}
```

When it receives a sendNotify message from the main subject program, all it has to do is to add the color name to the list.

```
public void sendNotify(string message){
       lsColors.Items.Add(message);
}
```

Our color window is also an observer, and it has to change the background color of the picture box and paint the color name using a brush. Note that we change the picture box's background color in the sendNotify event, and change the text in a paint event. The entire class is shown here.

```
public class ColObserver : Form, Observer{
       private Container components = null;
       private Brush bBrush;
       private System.Windows.Forms.PictureBox pic;
       private Font fnt;
```

Copyright © , 2002 by James W Cooper

```
private Hashtable colors;
private string colName;
//-----
public ColObserver(Subject subj)        {
       InitializeComponent();
       init(subj);
}
//-----
private void init(Subject subj) {
       subj.registerInterest (this);
       fnt = new Font("arial", 18, FontStyle.Bold);
       bBrush = new SolidBrush(Color.Black);
       pic.Paint+= new PaintEventHandler (paintHandler);
       //make Hashtable for converting color strings
       colors = new Hashtable ();
       colors.Add("red", Color.Red );
       colors.Add ("blue", Color.Blue );
       colors.Add ("green", Color.Green );
       colName = "";
}
//-----
public void sendNotify(string message) {
       colName = message;
       message = message.ToLower ();
       //convert color string to color object
       Color col = (Color)colors[message];
       pic.BackColor = col;
}
//-----
private void paintHandler(object sender,
             PaintEventArgs e) {
        Graphics g = e.Graphics ;
        g.DrawString(colName, fnt, bBrush, 20, 40)
}
```

Note that our sendNotify event receives a string representing the color name, and that we use a Hashtable to convert these strings to actual Color objects.

Meanwhile, in our main program, every time someone clicks on one of the radio buttons, it calls the *sendNotify* method of each Observer who has registered interest in these changes by simply running through the objects in the Observer's Collection.

```
private void opButton_Click(object sender, EventArgs e) {
       RadioButton but = (RadioButton)sender;
       for(int i=0; i< observers.Count ; i++ ) {
              Observer obs = (Observer)observers[i];
              obs.sendNotify (but.Text );
```

Copyright © , 2002 by James W Cooper

```
        }
}
```

In the case of the ColorForm observer, the sendNotify method changes the background color and the text string in the form Picturebox. In the case of the ListForm observer, however, it just adds the name of the new color to the list box. We see the final program running in Figure 27-3

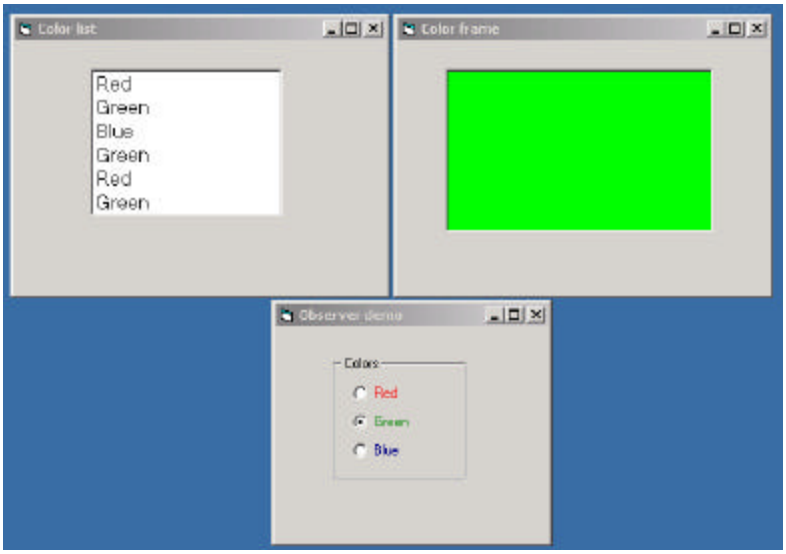


Figure 27-3 – The data control panel generates data that is displayed simultaneously as a colored panel and as a list box. This is a candidate for an Observer pattern.

The Message to the Media

Now, what kind of notification should a subject send to its observers? In this carefully circumscribed example, the notification message is the string representing the color itself. When we click on one of the radio buttons, we can get the caption for that button and send it to the observers. This, of course, assumes that all the observers can handle that string representation. In more realistic situations, this might not always be the case, especially if the observers could also be used to observe other data objects. Here we undertake two simple data conversions.

Copyright © , 2002 by James W Cooper

1. We get the label from the radio button and send it to the observers.
2. We convert the label to an actual color in the ColObserver.

In more complicated systems, we might have observers that demand specific, but different, kinds of data. Rather than have each observer convert the message to the right data type, we could use an intermediate Adapter class to perform this conversion.

Another problem observers may have to deal with is the case where the data of the central subject class can change in several ways. We could delete points from a list of data, edit their values, or change the scale of the data we are viewing. In these cases we either need to send different change messages to the observers or send a single message and then have the observer ask which sort of change has occurred.

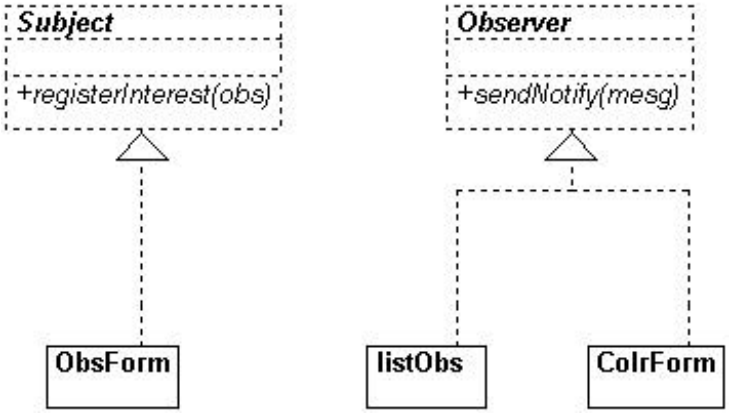


Figure 28-4 – The Observer interface and Subject interface implementation of the Observer pattern

Consequences of the Observer Pattern

Observers promote abstract coupling to Subjects. A subject doesn't know the details of any of its observers. However, this has the potential disadvantage of successive or repeated updates to the Observers when there are a series of incremental changes to the data. If the cost of these updates is high, it may be necessary to introduce some sort of change management so the Observers are not notified too soon or too frequently.

Copyright © , 2002 by James W Cooper

When one client makes a change in the underlying data, you need to decide which object will initiate the notification of the change to the other observers. If the Subject notifies all the observers when it is changed, each client is not responsible for remembering to initiate the notification. On the other hand, this can result in a number of small successive updates being triggered. If the clients tell the Subject when to notify the other clients, this cascading notification can be avoided, but the clients are left with the responsibility of telling the Subject when to send the notifications. If one client "forgets," the program simply won't work properly.

Finally, you can specify the kind of notification you choose to send by defining a number of update methods for the Observers to receive, depending on the type or scope of change. In some cases, the clients will thus be able to ignore some of these notifications.

Programs on the CD-ROM

`\Observer`	Observer example

Copyright © , 2002 by James W Cooper

28. The State Pattern

The State pattern is used when you want to have an object represent the state of your application and switch application states by switching objects. For example, you could have an enclosing class switch between a number of related contained classes and pass method calls on to the current contained class. *Design Patterns* suggests that the State pattern switches between internal classes in such a way that the enclosing object appears to change its class. In C#, at least, this is a bit of an exaggeration, but the actual purpose to which the classes are applied can change significantly.

Many programmers have had the experience of creating a class that performs slightly different computations or displays different information based on the arguments passed into the class. This frequently leads to some types of *select case* or *if-else* statements inside the class that determine which behavior to carry out. It is this inelegance that the State pattern seeks to replace.

Sample Code

Let's consider the case of a drawing program similar to the one we developed for the Memento class. Our program will have toolbar buttons for Select, Rectangle, Fill, Circle, and Clear. We show this program in Figure 28-1

Copyright © , 2002 by James W Cooper

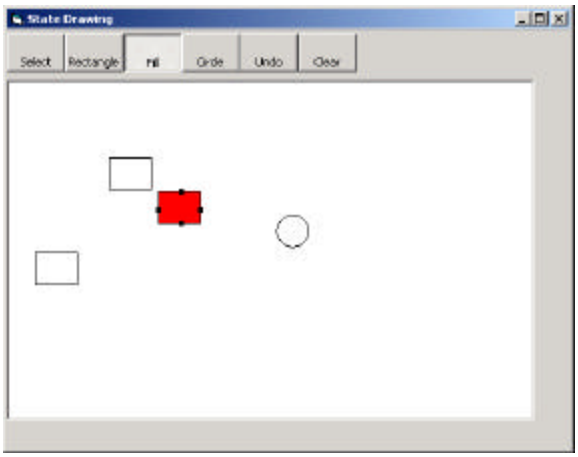


Figure 28-1 – A simple drawing program we will use for illustrating the State pattern

Each one of the tool buttons does something rather different when it is selected and you click or drag your mouse across the screen. Thus, the *state* of the graphical editor affects the behavior the program should exhibit. This suggests some sort of design using the State pattern.

Initially we might design our program like this, with a Mediator managing the actions of five command buttons, as shown in Figure 28-2

Copyright © , 2002 by James W Cooper

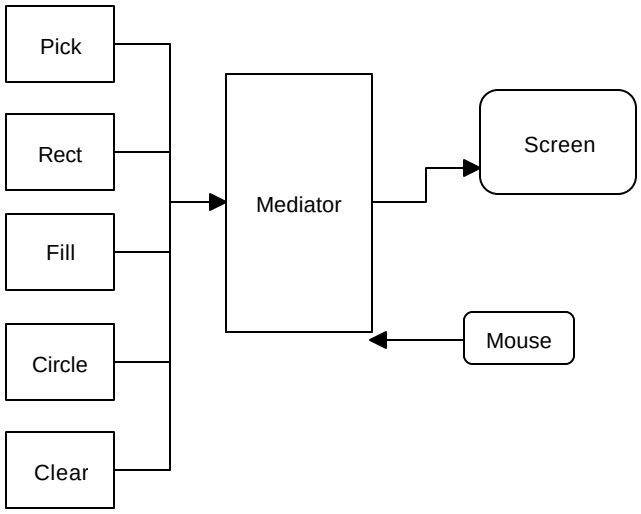


Figure 28-2– One possible interaction between the classes needed to support the simple drawing program

However, this initial design puts the entire burden of maintaining the state of the program on the Mediator, and we know that the main purpose of a Mediator is to coordinate activities between various controls, such as the buttons. Keeping the state of the buttons and the desired mouse activity inside the Mediator can make it unduly complicated, as well as leading to a set of *If* or S*elect* tests that make the program difficult to read and maintain.

Further, this set of large, monolithic conditional statements might have to be repeated for each action the Mediator interprets, such as mouseUp, mouseDrag, rightClick, and so forth. This makes the program very hard to read and maintain.

Instead, let's analyze the expected behavior for each of the buttons.

1. If the Select button is selected, clicking inside a drawing element should cause it to be highlighted or appear with

Copyright © , 2002 by James W Cooper

"handles." If the mouse is dragged and a drawing element is already selected, the element should move on the screen.

2. If the Rect button is selected, clicking on the screen should cause a new rectangle drawing element to be created.
3. If the Fill button is selected and a drawing element is already selected, that element should be filled with the current color. If no drawing is selected, then clicking inside a drawing should fill it with the current color.
4. If the Circle button is selected, clicking on the screen should cause a new circle drawing element to be created.
5. If the Clear button is selected, all the drawing elements are removed.

There are some common threads among several of these actions we should explore. Four of them use the mouse click event to cause actions. One uses the mouse drag event to cause an action. Thus, we really want to create a system that can help us redirect these events based on which button is currently selected.

Let's consider creating a State object that handles mouse activities.

```
public class State     {
       //keeps state of each button
       protected Mediator med;
       public State(Mediator md) {
              med = md;     //save reference to mediator
       }
       public virtual void mouseDown(int x, int y) {}
       public virtual void mouseUp(int x, int y) {    }
       public virtual void mouseDrag(int x, int y) {}
}
```

Note that we are creating an actual class here with empty methods, rather than an interface. This allows us to derive new State objects from this class and only have to fill in the mouse actions that actually do anything for that case. Then we'll create four derived State classes for Pick, Rect, Circle, and Fill and put instances of all of them inside a StateManager

Copyright © , 2002 by James W Cooper

class that sets the current state and executes methods on that state object. In *Design Patterns*, this StateManager class is referred to as a *Context*. This object is illustrated in Figure 28-3.

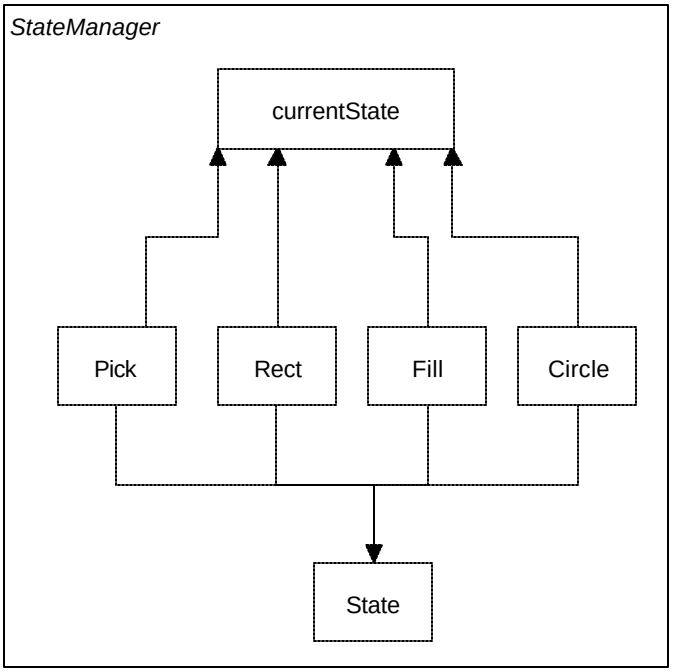


Figure 28-3– A StateManager class that keeps track of the current state

A typical State object simply overrides those event methods that it must handle specially. For example, this is the complete Rectangle state object. Note that since it only needs to respond to the mouseDown event, we don't have to write any code at all for the other events.

```
public class RectState :State    {
       public RectState(Mediator md):base (md) {}
       //-----
       public override void mouseDown(int x, int y) {
              VisRectangle vr = new VisRectangle(x, y);
              med.addDrawing (vr);
```

Copyright © , 2002 by James W Cooper

```
        }
}
```

The RectState object simply tells the Mediator to add a rectangle drawing to the drawing list. Similarly, the Circle state object tells the Mediator to add a circle to the drawing list.

```
public class CircleState : State {
        public CircleState(Mediator md):base (md){ }
        //-----
        public override void mouseDown(int x, int y) {
                VisCircle c = new VisCircle(x, y);
                med.addDrawing (c);
        }
}
```

The only tricky button is the Fill button because we have defined two actions for it.

1. If an object is already selected, fill it.
2. If the mouse is clicked inside an object, fill that one.

In order to carry out these tasks, we need to add the *selectOne* method to our base State interface. This method is called when each tool button is selected.

```
public class State     {
        //keeps state of each button
        protected Mediator med;
        public State(Mediator md) {
                med = md;     //save reference to mediator
        }
        public virtual void mouseDown(int x, int y) {}
        public virtual void mouseUp(int x, int y) {    }
        public virtual void mouseDrag(int x, int y) {}
        public virtual void selectOne(Drawing d) {}
}
```

The Drawing argument is either the currently selected Drawing or null if none is selected. In this simple program, we have arbitrarily set the fill color to red, so our Fill state class becomes the following.

Copyright © , 2002 by James W Cooper

```
public class FillState : State    {
       public FillState(Mediator md): base(md) { }
       //-----
       public override void mouseDown(int x, int y) {
              //Fill drawing if you click inside one
              int i = med.findDrawing(x, y);
              if (i >= 0) {
                     Drawing d = med.getDrawing(i);
                     d.setFill(true);  //fill drawing
               }
       }
       //-----
       public override void selectOne(Drawing d) {
              //fill drawing if selected
              d.setFill (true);
       }
}
```

Switching Between States

Now that we have defined how each state behaves when mouse events are sent to it, we need to examine how the StateManager switches between states. We create an instance of each state, and then we simply set the currentState variable to the state indicated by the button that is selected.

```
public class StateManager  {
       private State currentState;
       private RectState rState;
       private ArrowState aState;
       private CircleState cState;
       private FillState fState;

       public StateManager(Mediator med)              {
              //create an instance of each state
              rState = new RectState(med);
              cState = new CircleState(med);
              aState = new ArrowState(med);
              fState = new FillState(med);
              //and initialize them
              //set default state
              currentState = aState;
       }
```

Copyright © , 2002 by James W Cooper

Note that in this version of the StateManager, we create an instance of each state during the constructor and copy the correct one into the state variable when the set methods are called. It would also be possible to create these states on demand. This might be advisable if there are a large number of states that each consume a fair number of resources.

The remainder of the state manager code simply calls the methods of whichever state object is current. This is the critical piece—there is no conditional testing. Instead, the correct state is already in place, and its methods are ready to be called.

```
public void mouseDown(int x, int y) {
       currentState.mouseDown (x, y);
}
public void mouseUp(int x, int y) {
       currentState.mouseUp (x, y);
}
public void mouseDrag(int x, int y) {
       currentState.mouseDrag (x, y);
}
public void selectOne(Drawing d) {
       currentState.selectOne (d);
}
```

How the Mediator Interacts with the State Manager

We mentioned that it is clearer to separate the state management from the Mediator's button and mouse event management. The Mediator is the critical class, however, since it tells the StateManager when the current program state changes. The beginning part of the Mediator illustrates how this state change takes place. Note that each button click calls one of these methods and changes the state of the application. The remaining statements in each method simply turn off the other toggle buttons so only one button at a time can be depressed.

```
public class Mediator      {
       private bool startRect;
       private int selectedIndex;
```

Copyright © , 2002 by James W Cooper

```
private RectButton rectb;
private bool dSelected;
private ArrayList drawings;
private ArrayList undoList;
private RectButton rButton;
private FillButton filButton;
private CircleButton circButton;
private PickButton arrowButton;
private PictureBox canvas;
private int selectedDrawing;
private StateManager stMgr;
//-----
public Mediator(PictureBox pic)         {
       startRect = false;
       dSelected = false;
       drawings = new ArrayList();
       undoList = new ArrayList();
       stMgr = new StateManager(this);
       canvas = pic;
       selectedDrawing = -1;
}
//-----
public void startRectangle() {
       stMgr.setRect();
       arrowButton.setSelected(false);
       circButton.setSelected(false);
       filButton.setSelected(false);
}
//-----
public void startCircle() {
       stMgr.setCircle();
       rectb.setSelected(false);
       arrowButton.setSelected(false);
       filButton.setSelected(false);
}
```

The ComdToolBarButton

In the discussion of the Memento pattern, we created a series of button Command objects paralleling the toolbar buttons and keep them in a Hashtable to be called when the toolbar button click event occurs. However, a powerful alternative os to create a ComdToolBarButton class which implements the Command interface as

Copyright © , 2002 by James W Cooper

well as being a ToolBarButton. Then, each button can have an Execute method which defines its purpose. Here is the base class

```
public class ComdToolBarButton : ToolBarButton , Command    {
       private System.ComponentModel.Container components = null;
       protected Mediator med;
       protected  bool selected;
       public ComdToolBarButton(string caption, Mediator md)
       {
              InitializeComponent();
              med = md;
              this.Text =caption;
       }
       //------
       public void setSelected(bool b) {
              selected = b;
              if(!selected)
                     this.Pushed =false;
       }
       //-----
       public virtual void Execute() {
       }
```

Note that the Execute method is empty in this base class, but is virtual so we can override it in each derived class. In this case, we cannot use the IDE to create the toolbar, but can simply add the buttons to the toolbar programmatically:

```
       private void init() {
              //create a Mediator
              med = new Mediator(pic);
              //create the buttons
              rctButton = new RectButton(med);
              arowButton = new PickButton(med);
              circButton = new CircleButton(med);
              flButton = new FillButton(med);
              undoB = new UndoButton(med);
              clrb = new ClearButton(med);
              //add the buttons into the toolbar
              tBar.Buttons.Add(arowButton);
              tBar.Buttons.Add(rctButton);
              tBar.Buttons.Add(circButton);
              tBar.Buttons.Add(flButton);
              //include a separator
```

Copyright © , 2002 by James W Cooper

```
            ToolBarButton sep =new ToolBarButton();
            sep.Style = ToolBarButtonStyle.Separator;
            tBar.Buttons.Add(sep);
            tBar.Buttons.Add(undoB);
            tBar.Buttons.Add(clrb);
      }
```

Then we can catch all the toolbar button click events in a single method and call each button's Execute method.

```
private void tBar_ButtonClick(object sender,
                    ToolBarButtonClickEventArgs e) {
      Command comd = (Command)e.Button ;
      comd.Execute ();
}
```

The class diagram for this program illustrating the State pattern in this application is illustrated in two parts. The State section is shown in Figure 28-4

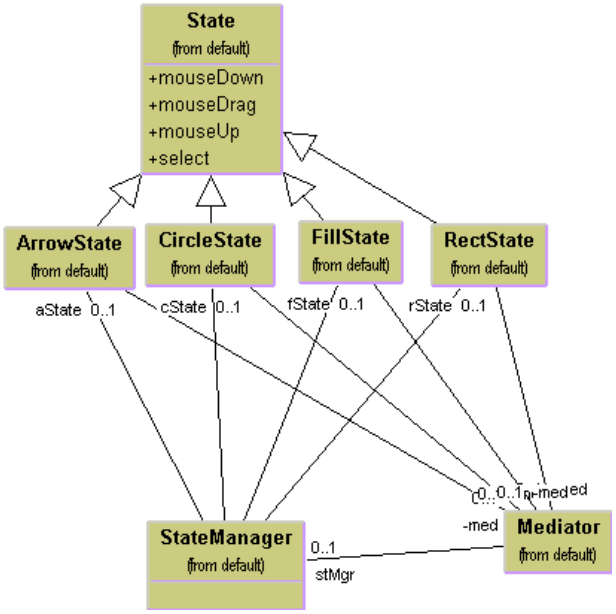


Figure 28-4 – The StateManager and the Mediator

Copyright © , 2002 by James W Cooper

The connection of the Mediator to the buttons is shown in Figure 28-5.

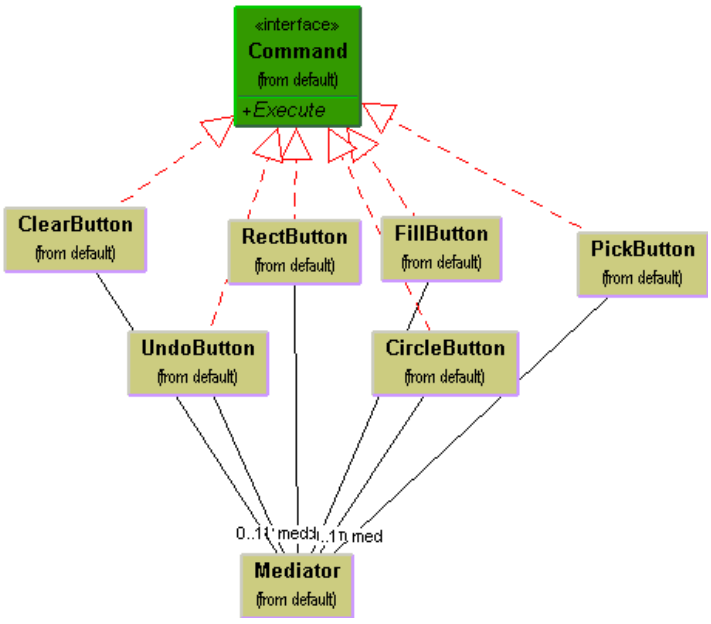


Figure 28-5 – Interaction between the buttons and the Mediator

Handling the Fill State

The Fill State object is only slightly more complex because we have to handle two cases. The program will fill the currently selected object if one exists or fill the next one that you click on. This means there are two State methods we have to fill in for these two cases, as we see here.

```
public class FillState : State   {
       public FillState(Mediator md): base(md) { }
       //-----
       public override void mouseDown(int x, int y) {
              //Fill drawing if you click inside one
              int i = med.findDrawing(x, y);
              if (i >= 0) {
                     Drawing d = med.getDrawing(i);
                     d.setFill(true);  //fill drawing
               }
```

Copyright © , 2002 by James W Cooper

```
        }
        //-----
        public override void selectOne(Drawing d) {
               //fill drawing if selected
               d.setFill (true);
        }
}
```

Handling the Undo List

Now we should be able to undo each of the actions we carry out in this drawing program, and this means that we keep them in an undo list of some kind. These are the actions we can carry out and undo.

1. Creating a rectangle
2. Creating a circle
3. Moving a rectangle or circle
4. Filling a rectangle or circle

In our discussion of the Memento pattern, we indicated that we would use a Memento object to store the state of the rectangle object and restore its position from that Memento as needed. This is generally true for both rectangles and circles, since we need to save and restore the same kind of position information. However, the addition of rectangles or circles and the filling of various figures are also activities we want to be able to undo. And, as we indicated in the previous Memento discussion, the idea of checking for the type of object in the undo list and performing the correct undo operation is a really terrible idea.

```
  //really terrible programming approach
      object obj = undoList[last];
      try{
             Memento mem = (Memento)obj;
             remove(mem);
      }
      catch (Exception) {
             removeDrawing();
      }
```

Copyright © , 2002 by James W Cooper

Instead, let's define the Memento as an interface.

```
public interface Memento   {
       void restore();
}
```

Then all of the objects we add into the undo list will implement the Memento interface and will have a restore method that performs some operation. Some kinds of Mementos will save and restore the coordinates of drawings, and others will simply remove drawings or undo fill states.

First, we will have both our circle and rectangle objects implement the Drawing interface.

```
public interface Drawing   {
       void setSelected(bool b);
       void draw(Graphics g);
       void move(int xpt, int ypt );
       bool contains(int x,int y);
       void setFill(bool b);
       CsharpPats.Rectangle getRects();
       void setRects(CsharpPats.Rectangle rect);
}
```

The Memento we will use for saving the state of a Drawing will be similar to the one we used in the Memento chapter, except that we specifically make it implement the Memento interface.

```
public class DrawMemento : Memento      {
       private int x, y, w, h;
       private Rectangle  rect;
       private Drawing visDraw;
       //------
       public DrawMemento(Drawing d)     {
              visDraw = d;
             rect = visDraw.getRects ();
             x = rect.x;
             y = rect.y ;
             w = rect.w;
             h = rect.h;
       }
       //-----
       public void restore() {
```

Copyright © , 2002 by James W Cooper

```
            //restore the state of a drawing object
            rect.x = x;
            rect.y = y;
            rect.h = h;
            rect.w = w;
            visDraw.setRects( rect);
      }
      }
```

Now for the case where we just want to remove a drawing from the list to be redrawn, we create a class to remember that index of that drawing and remove it when its *restore* method is called.

```
public class DrawInstance :Memento {
      private int intg;
      private Mediator med;
      //-----
      public DrawInstance(int intg, Mediator md)     {
            this.intg = intg;
            med = md;
      }
      //-----
      public int integ {
            get { return intg;               }
      }
      //-----
      public void restore() {
            med.removeDrawing(intg);
      }
}
```

We handle the FillMemento in just the same way, except that its restore method turns off the fill flag for that drawing element.

```
public class FillMemento : Memento      {
      private int index;
      private Mediator med;
      //-----
      public FillMemento(int dindex, Mediator md) {
            index = dindex;
            med = md;
      }
      //-----
      public void restore() {
```

Copyright © , 2002 by James W Cooper

```
            Drawing d = med.getDrawing(index);
            d.setFill(false);
        }
}
```

The VisRectangle and VisCircle Classes

We can take some useful advantage of inheritance in designing our visRectangle and visCircle classes. We make visRectangle *implement* the Drawing interface and then have visCircle *inherit* from visRectangle. This allows us to reuse the setSelected, setFill, and move methods and the rects properties. In addition, we can split off the drawHandle method and use it in both classes. Our new visRectangle class looks like this.

```
public class VisRectangle : Drawing     {
       protected int x, y, w, h;
       private const int SIZE=30;
       private CsharpPats.Rectangle rect;
       protected bool selected;
       protected bool filled;
       protected Pen bPen;
       protected SolidBrush bBrush, rBrush;
       //-----
       public VisRectangle(int xp, int yp)           {
              x = xp;                    y = yp;
              w = SIZE;                  h = SIZE;
              saveAsRect();
              bPen = new Pen(Color.Black);
              bBrush = new SolidBrush(Color.Black);
              rBrush = new SolidBrush (Color.Red );
       }
       //-----
       //used by Memento for saving and restoring state
        public CsharpPats.Rectangle getRects() {
              return rect;
       }
       //-----
        public void setRects(CsharpPats.Rectangle value) {
                    x=value.x;                y=value.y;
                    w=value.w;                h=value.h;
                    saveAsRect();
       }
```

Copyright © , 2002 by James W Cooper

```
//------
public void setSelected(bool b) {
       selected = b;
}
//-----
//move to new position
public void move(int xp, int yp) {
       x = xp;             y = yp;
       saveAsRect();
}
//-----
public virtual void draw(Graphics g) {
       //draw rectangle
       g.DrawRectangle(bPen, x, y, w, h);
       if(filled)
              g.FillRectangle (rBrush, x,y,w,h);
       drawHandles(g);
}
//-----
public void drawHandles(Graphics g) {
  if (selected) {   //draw handles
       g.FillRectangle(bBrush, x + w / 2, y - 2, 4, );
       g.FillRectangle(bBrush, x - 2, y + h / 2, 4, );
       g.FillRectangle(bBrush, x + (w / 2),
                     y + h - 2, 4, 4);
       g.FillRectangle(bBrush, x + (w - 2),
                     y + (h / 2), 4, 4);
  }
}
//-----
//return whether point is inside rectangle
public bool contains(int x, int y) {
       return rect.contains (x, y);
}
//------
//create Rectangle object from new position
protected void saveAsRect() {
       rect = new CsharpPats.Rectangle (x,y,w,h);
}
public void setFill(bool b) {
       filled = b;
}
```

Copyright © , 2002 by James W Cooper

However, our visCircle class only needs to override the draw method and have a slightly different constructor.

```
public class VisCircle : VisRectangle   {
       private int r;
       public VisCircle(int x, int y):base(x, y)            {
              r = 15; w = 30; h = 30;
              saveAsRect();
       }
       //-----
       public override void draw(Graphics g) {
              if (filled) {
                     g.FillEllipse(rBrush, x, y, w, h);
              }
       g.DrawEllipse(bPen, x, y, w, h);
       if (selected ){
              drawHandles(g);
              }
   }
}
```

Note that since we have made the x, y, and filled variables Protected, we can refer to them in the derived visCircle class without declaring them at all.

Mediators and the God Class

One real problem with programs with this many objects interacting is putting too much knowledge of the system into the Mediator so it becomes a "god class." In the preceding example, the Mediator communicates with the six buttons, the drawing list, and the StateManager. We could write this program another way so that the button Command objects communicate with the StateManager and the Mediator only deals with the buttons and the drawing list. Here, each button creates an instance of the required state and sends it to the StateManager. This we will leave as an exercise for the reader.

Copyright © , 2002 by James W Cooper

Consequences of the State Pattern

1. The State pattern creates a subclass of a basic State object for each state an application can have and switches between them as the application changes between states.
2. You don't need to have a long set of conditional *if* or *switch* statements associated with the various states, since each is encapsulated in a class.
3. Since there is no variable anywhere that specifies which state a program is in, this approach reduces errors caused by programmers forgetting to test this state variable
4. You could share state objects between several parts of an application, such as separate windows, as long as none of the state objects have specific instance variables. In this example, only the FillState class has an instance variable, and this could be easily rewritten to be an argument passed in each time.
5. This approach generates a number of small class objects but in the process simplifies and clarifies the program.
6. In C#, all of the States must implement a common interface, and they must thus all have common methods, although some of those methods can be empty. In other languages, the states can be implemented by function pointers with much less type checking and, of course, greater chance of error.

State Transitions

The transition between states can be specified internally or externally. In our example, the Mediator tells the StateManager when to switch between states. However, it is also possible that each state can decide automatically what each successor state will be. For example, when a rectangle or circle drawing object is created, the program could automatically switch back to the Arrow-object State.

Copyright © , 2002 by James W Cooper

Thought Questions

1. Rewrite the StateManager to use a Factory pattern to produce the states on demand.
2. While visual graphics programs provide obvious examples of State patterns, server programs can benefit by this approach. Outline a simple server that uses a state pattern.

Programs on the CD-ROM

`\State`	state drawing program

Copyright © , 2002 by James W Cooper

29. The Strategy Pattern

The Strategy pattern is much like the State pattern in outline but a little different in intent. The Strategy pattern consists of a number of related algorithms encapsulated in a driver class called the Context. Your client program can select one of these differing algorithms, or in some cases, the Context might select the best one for you. The intent is to make these algorithms interchangeable and provide a way to choose the most appropriate one. The difference between State and Strategy is that the user generally chooses which of several strategies to apply and that only one strategy at a time is likely to be instantiated and active within the Context class. By contrast, as we have seen, it is possible that all of the different States will be active at once, and switching may occur frequently between them. In addition, Strategy encapsulates several algorithms that do more or less the same thing, whereas State encapsulates related classes that each do something somewhat differently. Finally, the concept of transition between different states is completely missing in the Strategy pattern.

Motivation

A program that requires a particular service or function and that has several ways of carrying out that function is a candidate for the Strategy pattern. Programs choose between these algorithms based on computational efficiency or user choice. There can be any number of strategies, more can be added, and any of them can be changed at any time.

There are a number of cases in programs where we'd like to do the same thing in several different ways. Some of these are listed in the *Smalltalk Companion*.

- Save files in different formats.
- Compress files using different algorithms

Copyright © , 2002 by James W Cooper

- Capture video data using different compression schemes.
- Use different line-breaking strategies to display text data.
- Plot the same data in different formats: line graph, bar chart, or pie chart.

In each case we could imagine the client program telling a driver module (Context) which of these strategies to use and then asking it to carry out the operation.

The idea behind Strategy is to encapsulate the various strategies in a single module and provide a simple interface to allow choice between these strategies. Each of them should have the same programming interface, although they need not all be members of the same class hierarchy. However, they do have to implement the same programming interface.

Sample Code

Let's consider a simplified graphing program that can present data as a line graph or a bar chart. We'll start with an abstract PlotStrategy class and derive the two plotting classes from it, as illustrated in Figure 29-1.

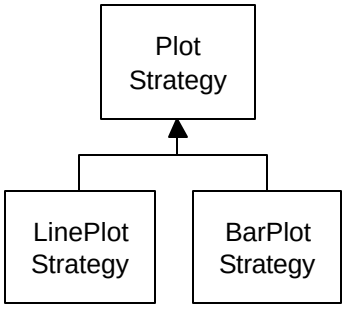


Figure 29-1 – Two instance of a PlotStrategy class

Our base PlotStrategy class is an abstract class containing the plot routine to be filled in in the derived strategy classes. It also contains the max and

Copyright © , 2002 by James W Cooper

min computation code, which we will use in the derived classes by containing an instance of this class.

```
public abstract class PlotStrategy      {
       public abstract void plot( float[] x, float[] y);
}
```

Then of the derived classes must implement a method called *plot* with two float arrays as arguments. Each of these classes can do any kind of plot that is appropriate.

The Context

The Context class is the traffic cop that decides which strategy is to be called. The decision is usually based on a request from the client program, and all that the Context needs to do is to set a variable to refer to one concrete strategy or another.

```
public class Context       {
       float[] x, y;
       PlotStrategy plts;  //strategy selected goes here
       //-----
       public void plot() {
              readFile();          //read in data
              plts.plot (x, y);
       }
       //-----
       //select bar plot
       public void setBarPlot() {
              plts = new BarPlotStrategy ();
       }
       //-----
       //select line plot
       public void setLinePlot() {
              plts = new LinePlotStrategy();
       }
       //-----
       public void readFile()           {
       //reads data in from data file

       }
}
```

Copyright © , 2002 by James W Cooper

The Context class is also responsible for handling the data. Either it obtains the data from a file or database or it is passed in when the Context is created. Depending on the magnitude of the data, it can either be passed on to the plot strategies or the Context can pass an instance of itself into the plot strategies and provide a public method to fetch the data.

The Program Commands

This simple program (Figure 29-2) is just a panel with two buttons that call the two plots. Each of the buttons is a derived button class the implements the Command interface. It selects the correct strategy and then calls the Context's plot routine. For example, here is the complete Line graph command button class.

Figure 29-2 – A simple panel to call two different plots

```
public class LineButton : System.Windows.Forms.Button, Command
{
	private System.ComponentModel.Container components = null;
	private Context contxt;

	public LineButton()		{
		InitializeComponent();
		this.Text = "Line plot";
	}
	public void setContext(Context ctx) {
		contxt = ctx;
	}
	public void Execute() {
		contxt.setLinePlot();
		contxt.plot();
	}
```

Copyright © , 2002 by James W Cooper

The Line and Bar Graph Strategies

The two strategy classes are pretty much the same: They set up the window size for plotting and call a plot method specific for that display panel. Here is the Line plot Strategy.

```
public class LinePlotStrategy : PlotStrategy   {
       public override void plot(float[] x, float[] y) {
              LinePlot lplt = new LinePlot();
              lplt.Show ();
              lplt.plot (x, y);
       }
}
```

The BarPlotStrategy is more or less identical.

The plotting amounts to copying in a reference to the x and y arrays, calling the scaling routine and then causing the Picturebox control to be refreshed, which will then call the paint routine to paint the bars.

```
public void plot(float[] xp, float[] yp) {
       x = xp;
       y = yp;
       setPlotBounds();    //compute scaling factors
       hasData = true;
       pic.Refresh();
}
```

Drawing Plots in C#

Note that both the LinePlot and the BarPlot window have plot methods that are called by the plot methods of the LinePlotStrategy and BarPlotStrategy classes. Both plot windows have a setBounds method that computes the scaling between the window coordinates and the x-y coordinate scheme. Since they can use the same scaling function, we write it once in the BarPlot window and derive the LinePlot window from it to use the same methods.

```
public virtual void setPlotBounds() {
       findBounds();
       //compute scaling factors
       h = pic.Height;
       w = pic.Width;
```

Copyright © , 2002 by James W Cooper

```
        xfactor = 0.8F * w / (xmax - xmin);
        xpmin = 0.05F * w;
        xpmax = w - xpmin;
        yfactor = 0.9F * h / (ymax - ymin);
        ypmin = 0.05F * h;
        ypmax = h - ypmin;
        //create array of colors for bars
        colors = new ArrayList();
        colors.Add(new SolidBrush(Color.Red));
        colors.Add(new SolidBrush(Color.Green));
        colors.Add(new SolidBrush(Color.Blue));
        colors.Add(new SolidBrush(Color.Magenta));
        colors.Add(new SolidBrush(Color.Yellow));
}
//-----
public int calcx(float xp) {
        int ix = (int)((xp - xmin) * xfactor + xpmin);
        return ix;
}
//-----
public int calcy(float yp) {
        yp = ((yp - ymin) * yfactor);
        int iy = h - (int)(ypmax - yp);
        return iy;
}
```

Making Bar Plots

The actual bar plot is drawn in a Paint routine that is called when a paint event occurs.

```
protected virtual void pic_Paint(object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;
        if (hasData) {
        for (int i = 0; i< x.Length; i++){
              int ix = calcx(x[i]);
              int iy = calcy(y[i]);
              Brush br = (Brush)colors[i];
               g.FillRectangle(br, ix, h - iy, 20, iy);
        }
}
```

Copyright © , 2002 by James W Cooper

Making Line Plots

The LinePlot class is very simple, since we derive it from the BarPlot class, and we need only write a new Paint method:

```
public class LinePlot :BarPlot {
       public LinePlot() {
              bPen = new Pen(Color.Black);
              this.Text = "Line Plot";
       }
       protected override void pic_Paint(object sender,
                     PaintEventArgs e) {
              Graphics g= e.Graphics;
              if (hasData) {
                     for (int i = 1; i< x.Length; i++) {
                            int ix = calcx(x[i - 1]);
                            int iy = calcy(y[i - 1]);
                            int ix1 = calcx(x[i]);
                            int iy1 = calcy(y[i]);
                            g.DrawLine(bPen, ix, iy, ix1, iy1);
                     }
              }
       }
}
```

The UML diagram showing these class relations is shown in Figure 29-3

Copyright © , 2002 by James W Cooper

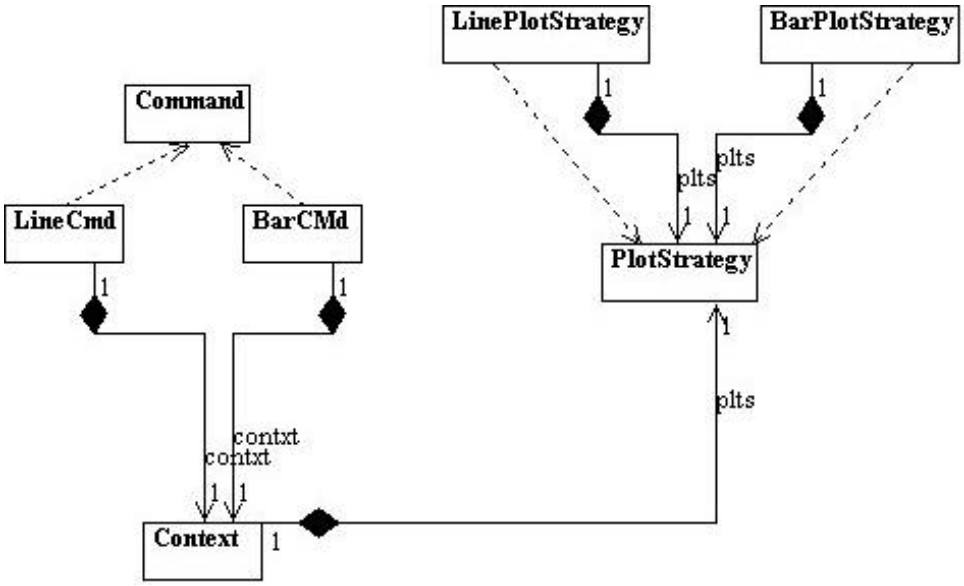
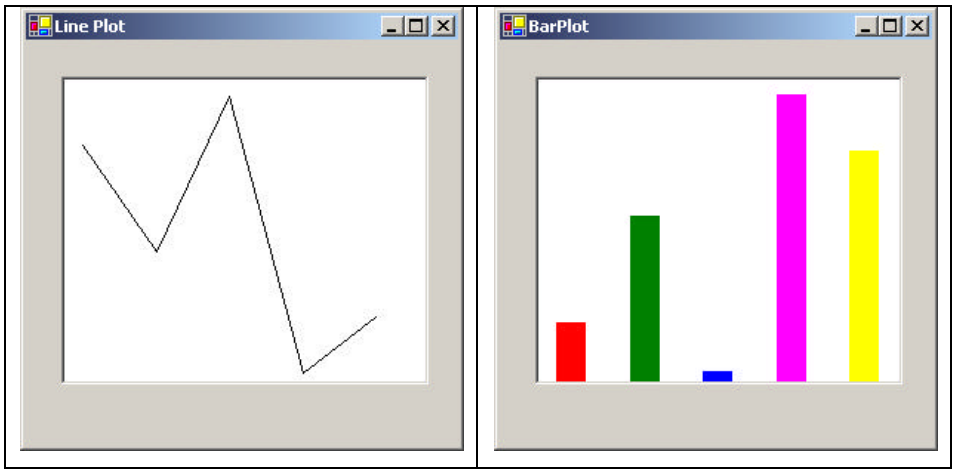


Figure 29-3 – The UML Diagram for the Strategy pattern

The final two plots are shown in Figure 29-4.



Copyright © , 2002 by James W Cooper

Figure 29-4– The line graph (left) and the bar graph (right)

Consequences of the Strategy Pattern

Strategy allows you to select one of several algorithms dynamically. These algorithms can be related in an inheritance hierarchy, or they can be unrelated as long as they implement a common interface. Since the Context switches between strategies at your request, you have more flexibility than if you simply called the desired derived class. This approach also avoids the sort of condition statements that can make code hard to read and maintain.

On the other hand, strategies don't hide everything. The client code is usually aware that there are a number of alternative strategies, and it has some criteria for choosing among them. This shifts an algorithmic decision to the client programmer or the user.

Since there are a number of different parameters that you might pass to different algorithms, you have to develop a Context interface and strategy methods that are broad enough to allow for passing in parameters that are not used by that particular algorithm. For example the *setPenColor* method in our PlotStrategy is actually only used by the LineGraph strategy. It is ignored by the BarGraph strategy, since it sets up its own list of colors for the successive bars it draws.

Programs on the CD-ROM

`\Strategy`	plot strategy

Copyright © , 2002 by James W Cooper

30. The Template Method Pattern

The Template Method pattern is a very simple pattern that you will find yourself using frequently. Whenever you write a parent class where you leave one or more of the methods to be implemented by derived classes, you are in essence using the Template pattern. The Template pattern formalizes the idea of defining an algorithm in a class but leaving some of the details to be implemented in subclasses. In other words, if your base class is an abstract class, as often happens in these design patterns, you are using a simple form of the Template pattern.

Motivation

Templates are so fundamental, you have probably used them dozens of times without even thinking about it. The idea behind the Template pattern is that some parts of an algorithm are well defined and can be implemented in the base class, whereas other parts may have several implementations and are best left to derived classes. Another main theme is recognizing that there are some basic parts of a class that can be factored out and put in a base class so they do not need to be repeated in several subclasses.

For example, in developing the BarPlot and LinePlot classes we used in the Strategy pattern examples in the previous chapter, we discovered that in plotting both line graphs and bar charts we needed similar code to scale the data and compute the x and y pixel positions.

```
public abstract class PlotWindow : Form {
       protected float ymin, ymax, xfactor, yfactor;
       protected float xpmin, xpmax, ypmin, ypmax, xp, yp;
       private float xmin, xmax;
       protected int w, h;
       protected float[] x, y;
       protected Pen bPen;
       protected bool hasData;
       protected const float max = 1.0e38f;
       protected PictureBox pic;
       //-----
```

Copyright © , 2002 by James W Cooper

```
protected virtual void init() {
       pic.Paint += new PaintEventHandler (pic_Paint);
}
//-----
public void setPenColor(Color c){
       bPen = new Pen(c);
}
//-----
public void plot(float[] xp, float[] yp) {
       x = xp;
       y = yp;
       setPlotBounds();    //compute scaling factors
       hasData = true;
}
//-----
public void findBounds() {
       xmin = max;
       xmax = -max;
       ymin = max;
       ymax = -max;
       for (int i = 0; i<  x.Length ; i++) {
              if (x[i] > xmax) xmax = x[i];
              if (x[i] < xmin) xmin = x[i];
              if (y[i] > ymax) ymax = y[i];
              if (y[i] < ymin) ymin = y[i];
       }
}
//-----
public virtual void setPlotBounds() {
       findBounds();
       //compute scaling factors
       h = pic.Height;
       w = pic.Width;
       xfactor = 0.8F * w / (xmax - xmin);
       xpmin = 0.05F * w;
       xpmax = w - xpmin;
       yfactor = 0.9F * h / (ymax - ymin);
       ypmin = 0.05F * h;
       ypmax = h - ypmin;
}
//-----
public int calcx(float xp) {
       int ix = (int)((xp - xmin) * xfactor + xpmin);
       return ix;
}
```

Copyright © , 2002 by James W Cooper

```
        //-----
        public int calcy(float yp) {
               yp = ((yp - ymin) * yfactor);
               int iy = h - (int)(ypmax - yp);
               return iy;
        }
        //-----
        public abstract void repaint(Graphics g) ;
        //-----
        protected virtual void pic_Paint(object sender,
                               PaintEvntArgs e) {
               Graphics g = e.Graphics;
               repaint(g);
        }
}
```

Thus, these methods all belong in a base PlotPanel class without any actual plotting capabilities. Note that the pic_Paint event handler just calls the abstract *repaint* method. The actual repaint method is deferred to the derived classes. It is exactly this sort of extension to derived classes that exemplifies the Template Method pattern.

Kinds of Methods in a Template Class

As discussed in *Design Patterns*, the Template Method pattern has four kinds of methods that you can use in derived classes.

1. Complete methods that carry out some basic function that all the subclasses will want to use, such as *calcx* and *calcy* in the preceding example. These are called *Concrete methods.*
2. Methods that are not filled in at all and must be implemented in derived classes. In C#, you would declare these as *virtual* methods.
3. Methods that contain a default implementation of some operations but that may be overridden in derived classes. These are called *Hook* methods. Of course, this is somewhat arbitrary because in C# you can override any public or protected method in the derived class but Hook methods, however, are intended to be overridden, whereas Concrete methods are not.

Copyright © , 2002 by James W Cooper

4. Finally, a Template class may contain methods that themselves call any combination of abstract, hook, and concrete methods. These methods are not intended to be overridden but describe an algorithm without actually implementing its details. *Design Patterns* refers to these as Template methods.

Sample Code

Let's consider a simple program for drawing triangles on a screen. We'll start with an abstract Triangle class and then derive some special triangle types from it, as we see in Figure 30-1

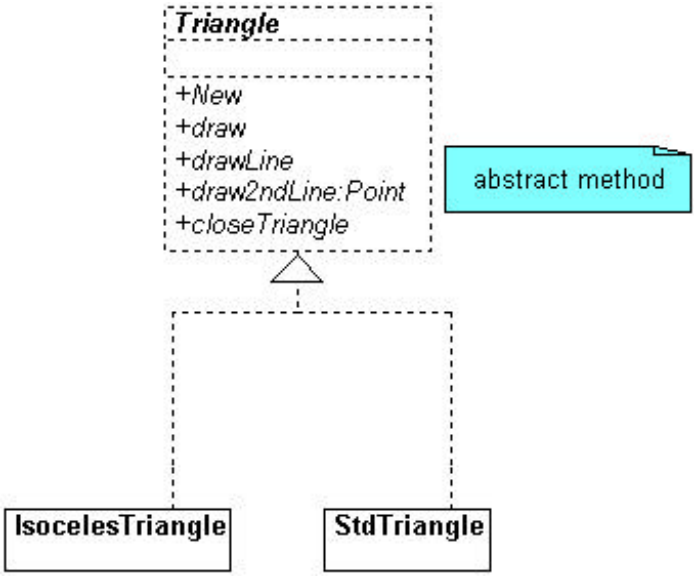


Figure 30-1 – The abstract Triangle class and three of its subclasses

Our abstract Triangle class illustrates the Template pattern.

```
public abstract class Triangle    {
       private Point p1, p2, p3;
       protected Pen pen;
       //-----
       public Triangle(Point a, Point b, Point c)     {
              p1 = a;
```

Copyright © , 2002 by James W Cooper

```
            p2 =  b;
            p3 = c;
            pen = new Pen(Color.Black , 1);
      }
      //-----
      public virtual void draw(Graphics g) {
            g.DrawLine (pen, p1, p2);
            Point c = draw2ndLine(g, p2, p3);
            closeTriangle(g, c);
      }
      //-----
      public abstract Point draw2ndLine(Graphics g,
                        Point a, Point b);
      //-----
      public void closeTriangle(Graphics g, Point c) {
            g.DrawLine (pen, c, p1);
      }
}
```

This Triangle class saves the coordinates of three lines, but the *draw* routine draws only the first and the last lines. The all-important *draw2ndLine* method that draws a line to the third point is left as an abstract method. That way the derived class can move the third point to create the kind of rectangle you wish to draw.

This is a general example of a class using the Template pattern. The *draw* method calls two concrete base class methods and one abstract method that must be overridden in any concrete class derived from Triangle.

Another very similar way to implement the triangle class is to include default code for the *draw2ndLine* method.

```
public virtual void draw2ndLine(Graphics g,
                  Point a, Point b) {
      g.drawLine(a, b);
}
```

In this case, the *draw2ndLine* method becomes a Hook method that can be overridden for other classes.

Copyright © , 2002 by James W Cooper

Drawing a Standard Triangle

To draw a general triangle with no restrictions on its shape, we simply implement the *draw2ndLine* method in a derived *stdTriangle* class.

```
public class StdTriangle :Triangle      {
       public StdTriangle(Point a, Point b, Point c)
                     : base(a, b, c) {}
       //------
       public override Point draw2ndLine(Graphics g,
                     Point a, Point b) {
              g.DrawLine (pen, a, b);
              return b;
       }
}
```

Drawing an Isosceles Triangle

This class computes a new third data point that will make the two sides equal in length and saves that new point inside the class.

```
public class IsocelesTriangle : Triangle       {
       private Point newc;
       private int newcx, newcy;
       //-----
       public IsocelesTriangle(Point a, Point b, Point c) :
                     base(a, b, c) {
              float dx1, dy1, dx2, dy2, side1, side2;
              float slope, intercept;
              int incr;
              dx1 = b.X - a.X;
              dy1 = b.Y - a.Y;
              dx2 = c.X  - b.X;
              dy2 = c.Y - b.Y;

              side1 = calcSide(dx1, dy1);
              side2 = calcSide(dx2, dy2);

              if (side2 < side1)
                     incr = -1;
              else
                     incr = 1;
              slope = dy2 / dx2;
              intercept = c.Y - slope * c.X;
```

Copyright © , 2002 by James W Cooper

```
            //move point c so that this is an isoceles triangle
            newcx = c.X;
            newcy = c.Y;
            while (Math.Abs (side1 - side2) > 1) {
                  //iterate a pixel at a time until close
                  newcx = newcx + incr;
                  newcy = (int)(slope * newcx + intercept);
                  dx2 = newcx - b.X;
                  dy2 = newcy - b.Y;
                  side2 = calcSide(dx2, dy2);
            }
            newc = new Point(newcx, newcy);
      }
      //-----
      private float calcSide(float a, float b) {
            return (float)Math.Sqrt (a*a +  b*b);
      }
}
```

When the Triangle class calls the *draw* method, it calls this new version of *draw2ndLine* and draws a line to the new third point. Further, it returns that new point to the *draw* method so it will draw the closing side of the triangle correctly.

```
      public override Point draw2ndLine(Graphics g,
                        Point b, Point  c) {
            g.DrawLine (pen, b, newc);
            return newc;
      }
```

The Triangle Drawing Program

The main program simply creates instances of the triangles you want to draw. Then it adds them to an ArrayList in the TriangleForm class.

```
private void init() {
      triangles = new ArrayList();
      StdTriangle t1 = new StdTriangle(new Point(10, 10),
            new Point(150, 50),
            new Point(100, 75));
      IsocelesTriangle t2 = new IsocelesTriangle(
            new Point(150, 100), new Point(240, 40),
            new Point(175, 150));
      triangles.Add(t1);
      triangles.Add(t2);
      Pic.Paint+= new PaintEventHandler (TPaint);
```

Copyright © , 2002 by James W Cooper

```
}
```

It is the *TPaint* method in this class that actually draws the triangles, by calling each Triangle's *draw* method.

```
private void TPaint (object sender,
              System.Windows.Forms.PaintEventArgs e) {
       Graphics g = e.Graphics;
       for (int i = 0; i<  triangles.Count ; i++) {
              Triangle t = (Triangle)triangles[i];
              t.draw(g);
       }
}
```

A standard triangle and an isosceles triangle are shown in Figure 30-2.

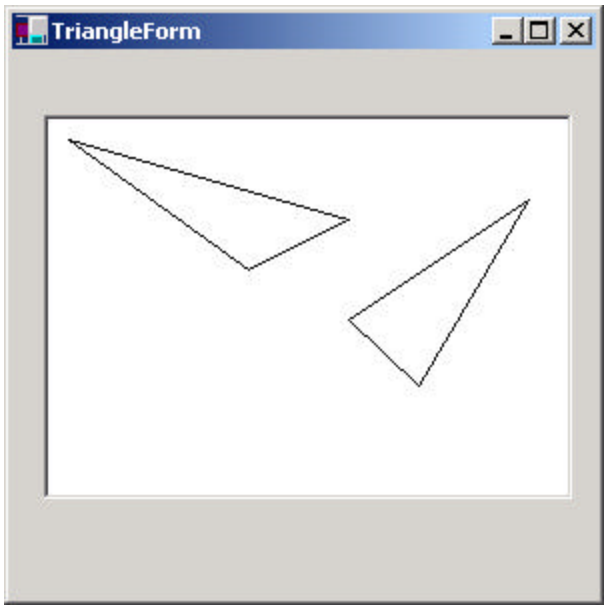


Figure 30-2 – A standard triangle and an isosceles triangle

Templates and Callbacks

Design Patterns points out that Templates can exemplify the "Hollywood Principle," or "Don't call us, we'll call you." The idea here is that methods

Copyright © , 2002 by James W Cooper

in the base class seem to call methods in the derived classes. The operative word here is *seem*. If we consider the *draw* code in our base Triangle class, we see that there are three method calls.

```
g.DrawLine (pen, p1, p2);
Point c = draw2ndLine(g, p2, p3);
closeTriangle(g, c);
```

Now *drawLine* and *closeTriangle* are implemented in the base class. However, as we have seen, the *draw2ndLine* method is not implemented at all in the base class, and various derived classes can implement it differently. Since the actual methods that are being called are in the derived classes, it appears as though they are being called from the base class.

If this idea makes you uncomfortable, you will probably take solace in recognizing that *all* the method calls originate from the derived class and that these calls move up the inheritance chain until they find the first class that implements them. If this class is the base class—fine. If not, it could be any other class in between. Now, when you call the *draw* method, the derived class moves up the inheritance tree until it finds an implementation of draw. Likewise, for each method called from within draw, the derived class starts at the current class and moves up the tree to find each method. When it gets to the *draw2ndLine* method, it finds it immediately in the current class. So it isn't "really" called from the base class, but it does seem that way.

Summary and Consequences

Template patterns occur all the time in OO software and are neither complex nor obscure in intent. They are a normal part of OO programming, and you shouldn't try to make them into more than they actually are.

The first significant point is that your base class may only define some of the methods it will be using, leaving the rest to be implemented in the derived classes. The second major point is that there may be methods in

Copyright © , 2002 by James W Cooper

the base class that call a sequence of methods, some implemented in the base class and some implemented in the derived class. This Template method defines a general algorithm, although the details may not be worked out completely in the base class.

Template classes will frequently have some abstract methods that you must override in the derived classes, and they may also have some classes with a simple "placeholder" implementation that you are free to override where this is appropriate. If these placeholder classes are called from another method in the base class, then we call these overridable methods "Hook" methods.

Programs on the CD-ROM

`\Template\Strategy`	plot strategy using Template method pattern
`\Template\Template`	plot of triangles

Copyright © , 2002 by James W Cooper

31. The Visitor Pattern

The Visitor pattern turns the tables on our object-oriented model and creates an external class to act on data in other classes. This is useful when you have a polymorphic operation that cannot reside in the class hierarchy for some reason—for example, because the operation wasn't considered when the hierarchy was designed or it would clutter the interface of the classes unnecessarily.

Motivation

While at first it may seem "unclean" to put operations inside one class that should be in another, there are good reasons for doing so. Suppose each of a number of drawing object classes has similar code for drawing itself. The drawing methods may be different, but they probably all use underlying utility functions that we might have to duplicate in each class. Further, a set of closely related functions is scattered throughout a number of different classes, as shown in Figure 31-1.

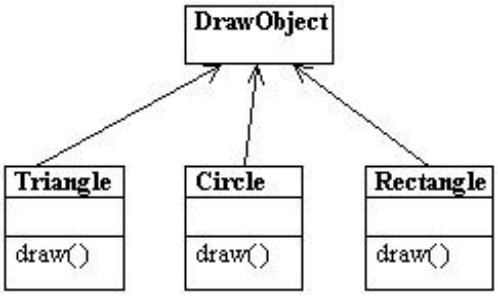


Figure 31-1 – A DrawObject and three of its subclasses

Instead, we write a Visitor class that contains all the related *draw* methods and have it visit each of the objects in succession (Figure 31-2).

Copyright © , 2002 by James W Cooper

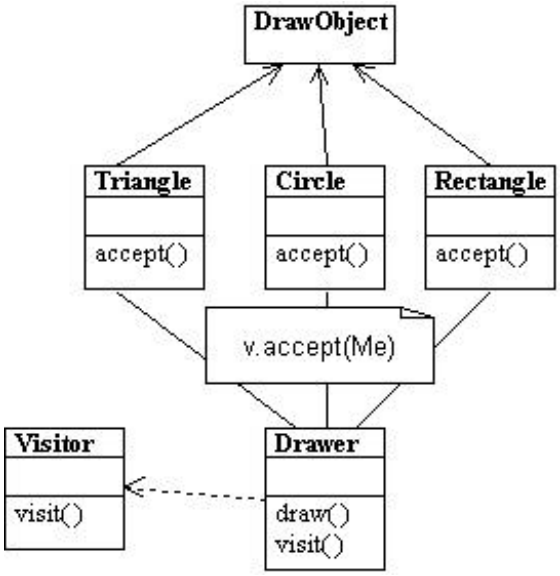


Figure 31-2 – A Visitor class (Drawer) that visits each of three triangle classes

The first question that most people ask about this pattern is "What does *visiting* mean?" There is only one way that an outside class can gain access to another class, and that is by calling its public methods. In the Visitor case, visiting each class means that you are calling a method already installed for this purpose, called *accept*. The *accept* method has one argument: the instance of the visitor. In return, it calls the *visit* method of the Visitor, passing itself as an argument, as shown in Figure 31-3.

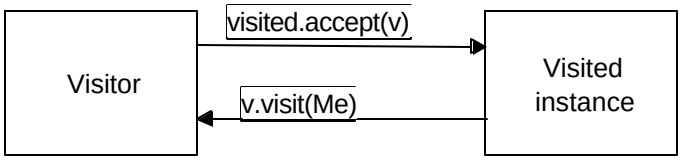


Figure 31-3 - How the visit and accept methods interact

Copyright © , 2002 by James W Cooper

Putting it in simple code terms, every object that you want to visit must have the following method.

```
public virtual void accept(Visitor v) {
            v.visit(this);
}
```

In this way, the Visitor object receives a reference to each of the instances, one by one, and can then call its public methods to obtain data, perform calculations, generate reports, or just draw the object on the screen. Of course, if the class does not have an *accept* method, you can subclass it and add one.

When to Use the Visitor Pattern

You should consider using a Visitor pattern when you want to perform an operation on the data contained in a number of objects that have different interfaces. Visitors are also valuable if you have to perform a number of unrelated operations on these classes. Visitors are a useful way to add function to class libraries or frameworks for which you either do not have the course or cannot change the source for other technical (or political) reasons. In these latter cases, you simply subclass the classes of the framework and add the *accept* method to each subclass.

On the other hand, as we will see, Visitors are a good choice only when you do not expect many new classes to be added to your program.

Sample Code

Let's consider a simple subset of the Employee problem we discussed in the Composite pattern. We have a simple Employee object that maintains a record of the employee's name, salary, vacation taken, and number of sick days taken. The following is a simple version of this class.

```
public class Employee     {
      int sickDays, vacDays;
      float salary;
      string name;
      public Employee(string name, float salary,
            int vDays, int sDays) {
      this.name = name;
      this.salary = salary;
```

Copyright © , 2002 by James W Cooper

```
        sickDays = sDays;
        vacDays = vDays;
}
//-----
public string getName() {
        return name;
}
public int getSickDays() {
        return sickDays;
}
public int getVacDays() {
        return vacDays;
}
public float getSalary() {
        return salary;
}
public virtual void accept(Visitor v) {
        v.visit(this);
}
}
```

Note that we have included the *accept* method in this class. Now let's suppose that we want to prepare a report on the number of vacation days that all employees have taken so far this year. We could just write some code in the client to sum the results of calls to each Employee's *getVacDays* function, or we could put this function into a Visitor.

Since C# is a strongly typed language, our base Visitor class needs to have a suitable abstract *visit* method for each kind of class in your program. In this first simple example, we only have Employees, so our basic abstract Visitor class is just the following.

```
public abstract class Visitor    {
              public abstract void visit(Employee emp);
              public abstract void visit(Boss bos);
}
```

Notice that there is no indication what the Visitor does with each class in either the client classes or the abstract Visitor class. We can, in fact, write a whole lot of visitors that do different things to the classes in our

Copyright © , 2002 by James W Cooper

program. The Visitor we are going to write first just sums the vacation data for all our employees.

```
public class VacationVisitor : Visitor  {
       private int totalDays;
       //-----
       public VacationVisitor()   {
              totalDays = 0;
       }
       //-----
       public int getTotalDays() {
              return totalDays;
       }
       //-----
       public override void visit(Employee emp){
              totalDays += emp.getVacDays ();
       }
       //-----
       public override void visit(Boss bos){
              totalDays += bos.getVacDays ();
       }
}
```

Visiting the Classes

Now all we have to do to compute the total vacation days taken is go through a list of the employees, visit each of them, and ask the Visitor for the total.

```
for (int i = 0;  i< empls.Length; i++) {
              empls[i].accept(vac);      //get the employee
}
lsVac.Items.Add("Total vacation days=" +
       vac.getTotalDays().ToString());
```

Let's reiterate what happens for each visit.

1. We move through a loop of all the Employees.
2. The Visitor calls each Employee's *accept* method.
3. That instance of Employee calls the Visitor's *visit* method.

Copyright © , 2002 by James W Cooper

4. The Visitor fetches the vacation days and adds them into the total.
5. The main program prints out the total when the loop is complete.

Visiting Several Classes

The Visitor becomes more useful when there are a number of different classes with different interfaces and we want to encapsulate how we get data from these classes. Let's extend our vacation days model by introducing a new Employee type called Boss. Let's further suppose that at this company, Bosses are rewarded with bonus vacation days (instead of money). So the Boss class has a couple of extra methods to set and obtain the bonus vacation day information.

```
public class Boss : Employee     {
       private int bonusDays;
       public Boss(string name, float salary,
              int vdays, int sdays):
                     base(name, salary, vdays, sdays)  { }
       public void setBonusDays(int bdays) {
              bonusDays = bdays;
       }
       public int getBonusDays() {
              return bonusDays;
       }
       public override void accept(Visitor v ) {
              v.visit(this);
       }
}
```

When we add a class to our program, we have to add it to our Visitor as well, so that the abstract template for the Visitor is now the following.

```
public abstract class Visitor    {
       public abstract void visit(Employee emp);
       public abstract void visit(Boss bos);
}
```

Copyright © , 2002 by James W Cooper

This says that any concrete Visitor classes we write must provide polymorphic *visit* methods for both the Employee class and the Boss class. In the case of our vacation day counter, we need to ask the Bosses for both regular and bonus days taken, so the visits are now different. We'll write a new bVacationVisitor class that takes account of this difference.

```
public class bVacationVisitor :Visitor  {
       private int totalDays;
       public bVacationVisitor()  {
              totalDays = 0;
       }
       //-----
       public override void visit(Employee emp) {
               totalDays += emp.getVacDays();
              try {
                     Manager mgr = (Manager)emp;
                     totalDays += mgr.getBonusDays();
              }
              catch(Exception ){}
       }
       //-----
       public override void visit(Boss bos) {
              totalDays += bos.getVacDays();
              totalDays += bos.getBonusDays();
       }
       //-----
       public int getTotalDays() {
              return totalDays;
       }
}
```

Note that while in this case Boss is derived from Employee, it need not be related at all as long as it has an *accept* method for the Visitor class. It is quite important, however, that you implement a *visit* method in the Visitor for *every class* you will be visiting and not count on inheriting this behavior, since the *visit* method from the parent class is an Employee rather than a Boss visit method. Likewise, each of your derived classes (Boss, Employee, etc.) must have its own *accept* method rather than calling one in its parent class. This is illustrated in the class diagram in Figure 31-4.

Copyright © , 2002 by James W Cooper

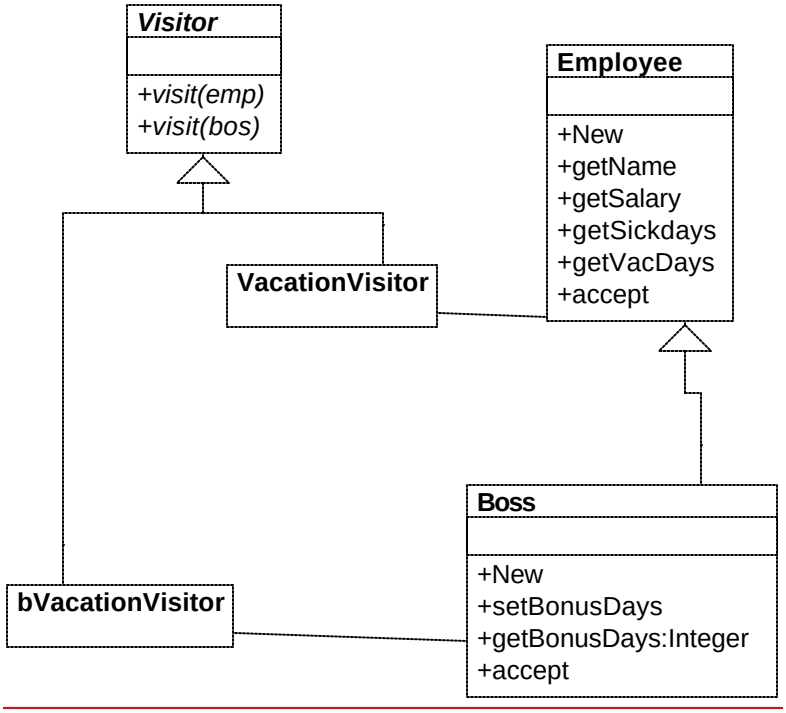


Figure 31-4 – The two visitor classes visiting the Boss and Employee classes

Bosses Are Employees, Too

We show in Figure 31-5 a simple application that carries out both Employee visits and Boss visits on the collection of Employees and Bosses. The original VacationVisitor will just treat Bosses as Employees and get only their ordinary vacation data. The bVacationVisitor will get both.

```
for (int i = 0;  i< empls.Length; i++) {
       empls[i].accept(vac);      //get the employee
       empls[i].accept(bvac);
}
lsVac.Items.Add("Total vacation days=" +
             vac.getTotalDays().ToString());
lsVac.Items.Add("Total boss vacation days=" +
```

Copyright © , 2002 by James W Cooper

```
                bvac.getTotalDays().ToString());
```

The two lines of displayed data represent the two sums that are computed when the user clicks on the Vacations button.

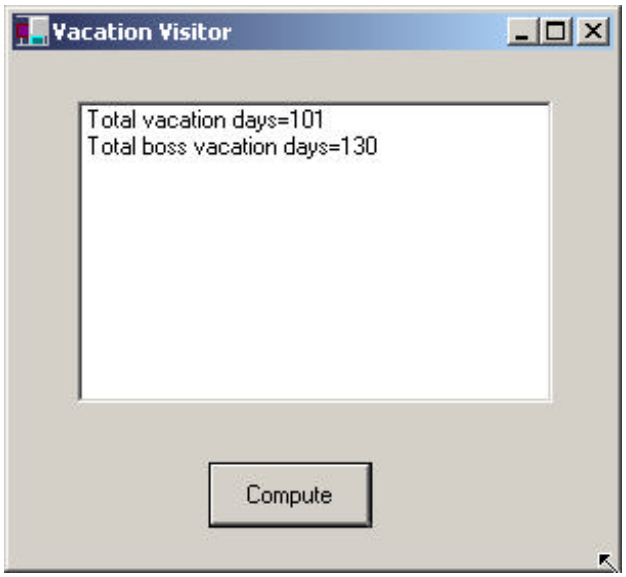


Figure 31-5 – A simple application that performs the vacation visits described

Catch-All Operations with Visitors

In the preceding cases, the Visitor class has a visit method for each visiting class, such as the following.

```
public abstract void visit(Employee emp);
public abstract void visit(Boss bos);
```

However, if you start subclassing your visitor classes and adding new classes that might visit, you should recognize that some *visit* methods might not be satisfied by the methods in the derived class. These might instead "fall through" to methods in one of the parent classes where that object type is recognized. This provides a way of specifying default visitor behavior.

Copyright © , 2002 by James W Cooper

Now every class must override *accept(v)* with its own implementation so the return call *v.visit(this)* returns an object *this* of the correct type and not of the superclass's type.

Let's suppose that we introduce another layer of management into our company: the Manager. Managers are subclasses of Employees, and now they have the privileges formerly reserved for Bosses of extra vacation days. Bosses now have an additional reward—stock options. Now if we run the same program to compute vacation days but do not revise our Visitor to look for Managers, it will recognize them as mere Employees and count only their regular vacation and not their extra vacation days. However, the catch-all parent class is a good thing if subclasses may be added to the application from time to time and you want the visitor operations to continue to run without modification.

There are three ways to integrate the new Manager class into the visitor system. You could define a ManagerVisitor or use the BossVisitor to handle both. However, there could be conditions when continually modifying the Visitor structure is not desirable. In that case, you could simply test for this special case in the EmployeeVisitor class.

```
public override void visit(Employee emp) {
       totalDays += emp.getVacDays();
      try {
            Manager mgr = (Manager)emp;
            totalDays += mgr.getBonusDays();
      }
      catch(Exception ){}
}
```

While this seems "unclean" at first compared to defining classes properly, it can provide a method of catching special cases in derived classes without writing whole new visitor program hierarchies. This "catch-all" approach is discussed in some detail in the book *Pattern Hatching* (Vlissides 1998).

Copyright © , 2002 by James W Cooper

Double Dispatching

No discussion on the Visitor pattern is complete without mentioning that you are really dispatching a method twice for the Visitor to work. The Visitor calls the polymorphic *accept* method of a given object, and the *accept* method calls the polymorphic *visit* method of the Visitor. It is this bidirectional calling that allows you to add more operations on any class that has an *accept* method, since each new Visitor class we write can carry out whatever operations we might think of using the data available in these classes.

Why Are We Doing This?

You may be asking yourself why we are jumping through these hoops when we could call the getVacationDays methods directly. By using this "callback" approach, we are implementing "double dispatching." There is no requirement that the objects we visit be of the same or even of related types. Further, using this callback approach, you can have a different visit method called in the Visitor, depending on the actual type of class. This is harder to implement directly.

Further, if the list of objects to be visited in an ArrayList is a collection of different types, having different versions of the visit methods in the actual Visitor is the only way to handle the problem without specifically checking the type of each class.

Traversing a Series of Classes

The calling program that passes the class instances to the Visitor must know about all the existing instances of classes to be visited and must keep them in a simple structure such as an array or collection. Another possibility would be to create an Enumeration of these classes and pass it to the Visitor. Finally, the Visitor itself could keep the list of objects that it is to visit. In our simple example program, we used an array of objects, but any of the other methods would work equally well.

Copyright © , 2002 by James W Cooper

Consequences of the Visitor Pattern

The Visitor pattern is useful when you want to encapsulate fetching data from a number of instances of several classes. *Design Patterns* suggests that the Visitor can provide additional functionality to a class without changing it. We prefer to say that a Visitor can add functionality to a collection of classes and encapsulate the methods it uses.

The Visitor is not magic, however, and cannot obtain private data from classes. It is limited to the data available from public methods. This might force you to provide public methods that you would otherwise not have provided. However, it can obtain data from a disparate collection of unrelated classes and utilize it to present the results of a global calculation to the user program.

It is easy to add new operations to a program using Visitors, since the Visitor contains the code instead of each of the individual classes. Further, Visitors can gather related operations into a single class rather than forcing you to change or derive classes to add these operations. This can make the program simpler to write and maintain.

Visitors are less helpful during a program's growth stage, since each time you add new classes that must be visited, you have to add an abstract *visit* operation to the abstract Visitor class, and you must add an implementation for that class to each concrete Visitor you have written. Visitors can be powerful additions when the program reaches the point where many new classes are unlikely.

Visitors can be used very effectively in Composite systems, and the boss-employee system we just illustrated could well be a Composite like the one we used in the Composite chapter.

Thought Question

An investment firm's customer records consist of an object for each stock or other financial instrument each investor owns. The object contains a history of the purchase, sale, and dividend activities for that stock. Design

Copyright © , 2002 by James W Cooper

a Visitor pattern to report on net end-of-year profit or loss on stocks sold during the year.

Programs on the CD-ROM

\Visitor\	Visitor example

Copyright © , 2002 by James W Cooper

32. Bibliography

Copyright © , 2002 by James W Cooper

Alexander, Christopher, Ishikawa, Sara, *et. al. A Pattern Language*, Oxford University Press, New York, 1977.

Alpert, S. R., Brown, K., and Woolf, B. *The Design Patterns Smalltalk Companion*, Addison-Wesley, Reading, MA, 1998.

Arnold, K., and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, MA, 1997.

Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *A System of Patterns*, John Wiley and Sons, New York, 1996.

Cooper, J. W. *Java Design Patterns: A Tutorial*. Addison-Wesley, Reading, MA, 2000.

Cooper, J. W. *Principles of Object-Oriented Programming in Java 1.1 Coriolis (Ventana), 1997.*

Cooper, J.W. *Visual Basic Design Patterns: VB6 and VB.NET*, Addison-Wesley, Boston, MA, 2001.

Coplien, James O. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

Coplien, James O., and Schmidt, Douglas C. *Pattern Languages of Program Design*, Addison-Wesley, Reading, MA, 1995.

Fowler, Martin, with Kendall Scott. *UML Distilled*, Addison-Wesley, Reading, MA, 1997.

Gamma, E., Helm, T., Johnson, R., and Vlissides, J. *Design Patterns: Abstraction and Reuse of Object Oriented Design.* Proceedings of ECOOP '93, 405—431.

Gamma, Eric, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns. Elements of Reusable Software*, Addison-Wesley, Reading, MA, 1995.

Grand, Mark *Patterns in Java*, Volume 1, John Wiley & Sons, New York 1998.

Krasner, G.E. and Pope, S.T. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programmng **I**(3)., 1988*

Kurata, Deborah, "Programming with Objects," *Visual Basic Programmer's Journal*, June, 1998.

Pree, Wolfgang, *Design Patterns for Object Oriented Software Development*, Addison-Wesley, 1994.

Riel, Arthur J., *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996

Vlissides, John, *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, Reading, MA, 1998

Copyright © , 2002 by James W Cooper

Copyright © , 2002 by James W Cooper